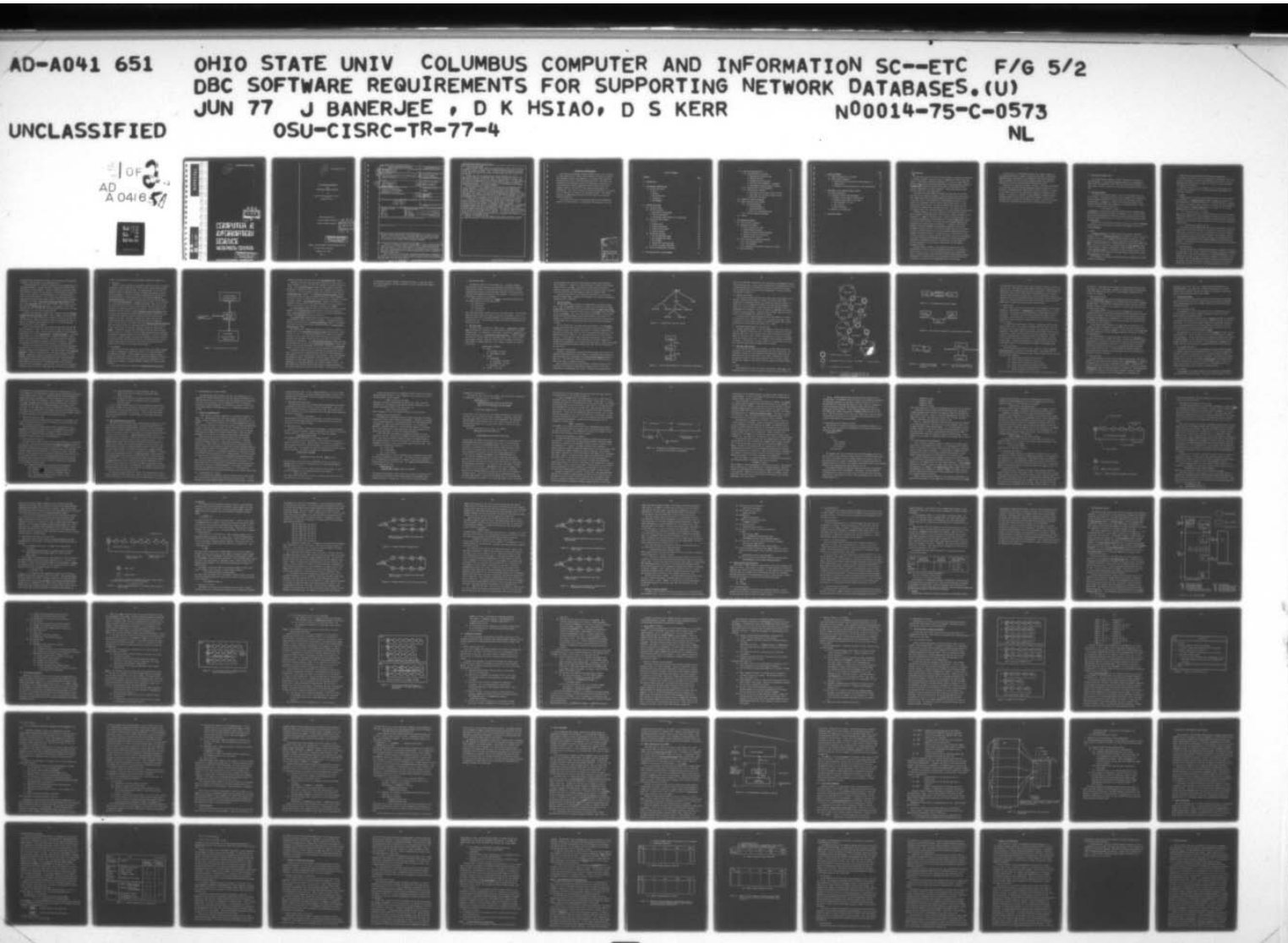
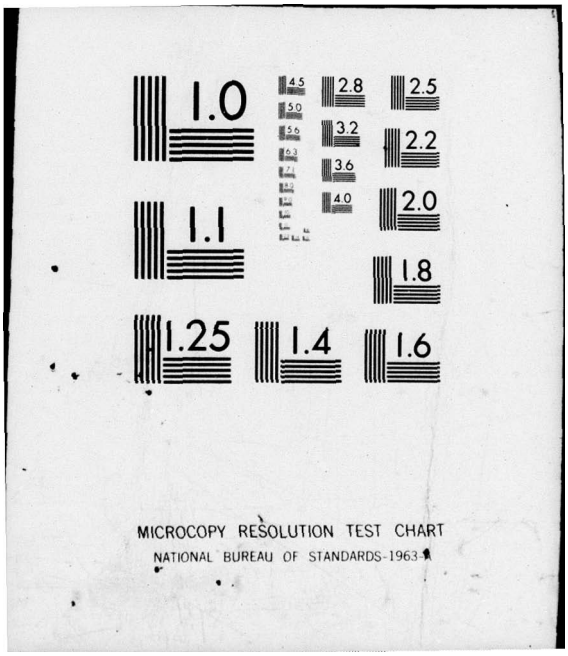
AD-A041 651 OHIO STATE UNIV COLUMBUS COMPUTER AND INFORMATION SC--ETC F/G 5/2
DBC SOFTWARE REQUIREMENTS FOR SUPPORTING NETWORK DATABASES.(U)
JUN 77 J BANERJEE , D K HSIAO, D S KERR N00014-75-C-0573
UNCLASSIFIED OSU-CISRC-TR-77-4 NL




1.0
4.5
5.0
5.6
6.3
2.8
3.2
3.6
4.0
2.5
2.2
2.0
1.8
1.1
1.25
1.4
1.6
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

TECHNICAL REPORT SERIES

COMPUTER & INFORMATION SCIENCE RESEARCH CENTER

THE OHIO STATE UNIVERSITY COLUMBUS, OHIO

(12)

(OSU-CISRC-TR-77-4)

DBC SOFTWARE REQUIREMENTS
FOR
SUPPORTING NETWORK DATABASES

by

Jayanta Banerjee, David K. Hsiao
and
Douglas S. Kerr

Work performed under

Contract N00014-75-C-0573

Office of Naval Research

Computer and Information Science Research Center

The Ohio State University

Columbus, OH 43210

June 77

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: OSU-CISRC-TR-77-4	2. GOVT ACCESSION NO.
3. RECIPIENT'S CATALOG NUMEER	
4. TITLE (and Subtitle): DBC Software Requirements for Supporting Network Databases.	5. TYPE OF REPORT & PERIOD COVERED: Technical Report.
	6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s): Jayanta Banerjee, David K. Hsiao, Douglas S. Kerr	8. CONTRACT OR GRANT NUMBER(s): N00014-75-C-0573
9. PERFORMING ORGANIZATION NAME AND ADDRESS: Office of Naval Research, Information Systems Program, Washington, D.C. 20360	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: 4115-A1
11. CONTROLLING OFFICE NAME AND ADDRESS	12. REPORT DATE: June 1977
	13. NUMBER OF PAGES: 91
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office): 99p.	15. SECURITY CLASS. (of this report)
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report) Approved for public release; distribution unlimited.

Scientific Officer	DDC New York Area	DISTRIBUTION STATEMENT A
ONR BRO	ONR 437	Approved for public release;
ACO	ONR, Boston	Distribution Unlimited
NRL 2627	ONR, Chicago	
ONR 102IP	ONR, Pasadena	

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)
Database Computer; data models; network data model; DBTG; CODASYL; keyword; attribute; attribute-value pair; clustering; directory; associative search; query; predicate; system performance; record type; record occurrence; set; set occurrence; area; schema; data manipulation language.

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This is the second of a series of three reports aimed at demonstrating the capabilities of a back-end database computer (DBC) in supporting known data models and systems. In the first report it was shown that the hierarchical data model can be supported on the DBC with a performance that is considerably better than existing hierarchical database systems such as IBM's Information Management System (IMS).

In this report we illustrate how a network data model can be supported on the DBC. The features of the network model are those outlined by the CODASYL

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

Data Base Task Group (DBTG).

Early sections of this report are introductory in nature. They have been included in order to familiarize the reader with important features of the DBC and of the network data model. Thus, a reader without any detailed knowledge of the DBC and the network data model can still study this report without much difficulty.

We then deal with the representation of DBTG information on the DBC. A DBC record is made up of elementary items called attribute-value pairs. Every DBTG record occurrence, on the other hand, is characterized by a record type, an area and its participation in an arbitrary number of sets. All these features of a record occurrence are captured in a DBC record by means of appropriate attribute-value pairs. For storing records in the DBC database, the automatic clustering facilities of the DBC are utilized.

We also show how user programs can be executed (via an interface module) without conversion. A user command written in the DBTG data manipulation language (DML) is intercepted by the interface module, which then generates required commands for the DBC and takes appropriate actions needed to execute the original user command. The main theme in designing the interface is to take advantage of the content-addressability of the DBC and the knowledge that a user program spends most of its time in traversing sets. We proceed then to discuss the interface buffer management and the running environment of the interface.

A comparison of the performances of the DBC and a conventional computer system in supporting a network database (namely, UNIVAC's DMS 1100) is given. Storage requirement and frequency of database accesses are analyzed. It is found that for the same database the secondary storage requirement of the DBC may be slightly higher than that of a conventional system. However, this increase in secondary storage is more than compensated by better response time, smaller directory memory and saving of the real memory. The number of database accesses can be as much as fifty times fewer on the DBC. The directory memory requirement on the DBC can be a hundredth of that required on a conventional system. In addition, the DBC provides new and advanced data management facilities which are not found in a conventional system.

Finally, we conclude with some remarks on the special features of the DBC and what other purpose it can serve besides supporting multiple data models.

PREFACE AND ACKNOWLEDGEMENT

This work was supported by Contract N00014-75-C-0573 from the Office of Naval Research to Dr. David K. Hsiao, Associate Professor of Computer and Information Science, and conducted at the Computer and Information Science Research Center of The Ohio State University. The Computer and Information Science Research Center of The Ohio State University is an interdisciplinary research organization which consists of the staff, graduate students, and faculty of many University departments and laboratories. This report is based on research accomplished in cooperation with the Department of Computer and Information Science. The research contract was administered and monitored by The Ohio State University Research Foundation.

The authors would like to thank Lorenzo Aguilar and Krishnamurthi Kannan for careful reading of the manuscripts and comments on the work.

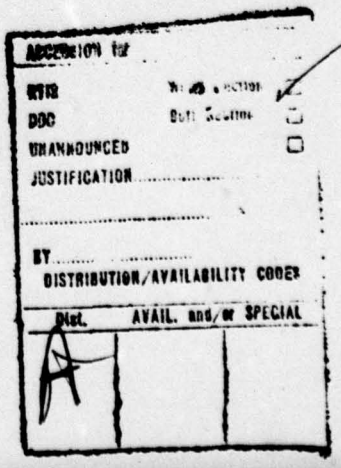


TABLE OF CONTENTS

1. INTRODUCTION

This is the second of a series of reports aimed at studying the capabilities of a database computer, known as the DBC, in supporting the three major data models: hierarchical, network and relational. In the first report [1], DBC software requirements for handling hierarchical databases have been presented. This report will be directed towards an investigation of the software requirements for network databases. Relational database systems will be treated in a forthcoming report.

The April 1971 report [2] of the CODASYL Data Base Task Group (DBTG) is chosen as the definitive document of network databases and systems since most commercially available network database management systems are based on the DBTG report. Even though many of these commercial systems use a syntax that is slightly different from the DBTG specifications, the main concepts have been retained. The network database model studied in this report will be referred to as the DBTG data model. Our emphasis will be on concepts rather than the detailed syntax of the DBTG language specifications.

Database computers are a recent addition to the family of computers. With the advent of large databases, there has been a growing awareness of the necessity of a computer architecture that is oriented towards storage, retrieval and manipulation of large quantities of information. The DBC [3,4,5] is a step in that direction. It utilizes content-addressable memories and processors with various speeds and capacities. In addition, it provides powerful clustering mechanisms for performance enhancement and security mechanisms for access control. The built-in hardware data structure enables the DBC to interface directly with existing database management application programs with minimal software. In other words, it is the purpose of this report to show that the required software is minimal and that the new software can replace existing database management systems with improved performance. Furthermore, the advanced features of the DBC allows the development of new user applications not now possible using present network database management systems.

The most salient construct in the network data model is the "set". Implementation of sets on the DBC is extremely efficient. The ordering of records within the sets is also considered important in the network data model. Although the DBC is not inherently oriented towards maintaining data in sequential or ordered fashion, this requirement will be satisfied since the DBC has the capability of transmitting data in a sorted order.

In this report, we will develop constructs that support features of the network data model in a manner that greatly outperforms "conventional" implementations. In particular, the linkage structures based on pointers used to represent sets on the regular secondary storage are replaced by pointer-free structures utilizing content-addressable storage.

This report is organized as follows: Sections 2 and 3 are introductions to the DBC and the network database management systems, repectively. In Section 4, we demonstrate how network data is stored in the DBC. In Section 5, we discuss how network data manipulation commands are converted to DBC queries. In Section 6, we propose a buffer management strategy required to interface between the DBC and user application programs. The report is concluded with an analysis of performance in Section 7 and with a discussion in Section 8 of additional features that can be supported on the DBC.

2. THE DATABASE COMPUTER (DBC)

As a special-purpose computer, the DBC is intended to be used as a back-end machine to a front-end conventional computer. It is designed to handle very large databases of 10^9 - 10^{10} bytes in an efficient manner. In this section, we shall concentrate on the major architectural features of the DBC.

2.1 The DBC Data Model

Let there be two primitive sets: a set AT of "attributes" and a set VA of "values". The meaning of the two sets is assumed to be understood and is left otherwise undefined in order to allow for the broadest possible interpretation. A record R is a subset of the Cartesian product AT $\times$ VA, with the restriction that every attribute in a record is distinct. Thus, R is a set of ordered pairs of the form:

<an attribute, a value> .

The keywords of a record (or a group of records) are those attribute-value pairs which characterize the record (or the group), i.e., those pairs that may be used to distinguish the record (or the record group) from all others. The other attribute-value pairs of a record, if any, are collectively called the record-body.

The set of all records which are stored in the DBC is called the database. The database may be partitioned into subsets called files, each with its unique file-name.

A. Query

A keyword predicate is a triple of the form <attribute, relational operator, value >. A relational operator is an element of the set $\{=,\neq,<,\leq,\geq,>\}$. A keyword $<A,V>$ is said to satisfy a keyword predicate $<A_p,O_p,V_p>$ if and only if $A=A_p$ and $V\ O_p\ V_p$, i.e., V and V_p are related by the operator O_p. A query is a Boolean expression of keyword predicates in disjunctive normal form. Thus, a query is a disjunction of conjuncts known as query conjuncts, where a query conjunct is simply a conjunction of keyword predicates. A record in a file satisfies a query if it satisfies at least one query conjunct in the query. The set of all records in a file that satisfy a query will be called the response set of the query.

As an example of the types of queries that may be recognized by the DBC, consider the following:

([DEPT='TOY']∧[SALARY<10000])∨([DEPT='BOOK']∧[SALARY>50000]).

If the above query refers to a file of employees of a department store, then it will be satisfied by records of the employees working either in the toy department and earning less than 10,000, or working in the book department and making more than 50,000.

Queries are used not only to retrieve a set of records among all the records in the database but also to specify protection requirements and clustering conditions.

B. Security

The DBC allows for security specifications based on the actual contents of the database. A database access or simply an access is the name of a DBC operation which transfers information to or from the database. Examples of accesses are retrieve, insert and delete. For every user of the database, the DBC maintains a database capability, which is simply a list of file sanctions whose entries are of the form:

$$(F, [Q_1,A_1], [Q_2,A_2],..., [Q_n,A_n])$$

where F is a file name, each Q_i is a query and each A_i is a set of accesses. The database capability of a user determines the records he can access. For example, for a user to be allowed to perform an access operation a on record R of file F, the following condition must hold for every (Q_i,A_i) in this file sanction for F:

$$(R \text{ satisfies } Q_i) \text{ implies } (a \in A_i)$$

This type of security specification is powerful and elegant. With this specification, not only can security be enforced in terms of record types or entire files, but security can also be facilitated at a much more detailed level based on the actual content of the records in the database. And since such a mechanism is directly provided in the DBC, it may be gainfully and conveniently incorporated into any database management system supported by the DBC. A more detailed and formal discussion of the DBC security provisions will be found in [3].

C. DBC Architecture

The most natural way of addressing information in a database is in terms of the content of the records. However, the secondary storages of conventional computers have so far been restricted only to location-addressability. This implies that in order to find a record in the database, the location of the record must first be determined via software techniques and auxiliary data structures. The overhead therefore includes the complexity of software to support auxiliary data structures. This overhead becomes particularly

intolerable when the database is large, since the search of the auxiliary structure itself becomes a time-consuming process.

The DBC provides for the entire database an on-line storage which can be content-addressed. Although associative memory also provides content-addressing, it is not possible to develop a monolithic associative memory with sufficient capacity for DBC storage. By partitioning the memory into blocks, each of which is content-addressable, and by limiting access to only one of these blocks at a time, the DBC can achieve some degree of associativity and very large storage capacity. Such a processor and memory organization is termed a partitioned content-addressable memory (PCAM). The on-line mass memory (MM) of the DBC is a PCAM. Each partition of the MM is called a minimal access unit (MAU). As an example, a 10^9 - byte database will have 1,000 MAUs each of which processes and stores 10^6 bytes, which happens to be the size of a disk cylinder.

Another major component of the DBC is a processor called the database command and control processor (DBCCP). When a command from the front-end computer (the one which interfaces with the user) is sent to the DBC, the DBCCP decodes the command, determines the MAUs to be searched in order to satisfy the command, issues appropriate orders to the MM and transfers data to/from the front-end computer.

Since a large database will contain many MAUs and since only one MAU can be accessed at a time, it is not practical to search all the MAUs for every search order. Hence, directory entries are made for certain keywords. These keywords are called Type-D keywords or directory keywords. A directory entry consists of a Type-D keyword and the numbers of the MAUs in which records containing this keyword appear. Any query conjunct is expected to have at least one predicate consisting of a directory keyword. Otherwise, an exhaustive search of the MM will be necessary to satisfy the query.

The directory entries also contain clustering and security information. The collection of all the directory entries is also stored in a PCAM with different capacity and processing speed. This PCAM is known as the structure memory(SM). Typically, directories are of the order of 1% to 10% of the database. Therefore, the SM has a capacity of 10^7 to 10^9 bytes. It is estimated that a query conjunct will seldom have more than 20 predicates; and a single MAU access will normally satisfy a query. Therefore, the access speed of the SM is about 1 millisecond which is about 20 times faster than the time required to access an MAU. Thus, in the time required to process a query in the SM, another query may be satisfied by accessing

an MAU. The relationship of SM, MM and DBCCP is depicted in Figure 2.1.

D. Clustering

Based on certain prespecified information created by the user, clustering of records is done automatically by the DBC, so that records being accessed together are stored in the same MAU. As a design policy, no two files are allowed to share the same MAU. The user is provided some degree of control over the placement of records in the MAUs by application of the concept of clustering keyword. Certain attributes of a file may be designated as clustering attributes. Keywords whose attributes are clustering attributes are termed clustering keywords. A cluster is then defined as a set of records all of which have the same set of clustering keywords. Each record in the file will then belong to one and only one cluster. The user may now impose two types of clustering conditions on the records. For each record to be inserted in the DBC, there is a single mandatory clustering condition, which is a query consisting of clustering keywords, that must be satisfied by one or more records existing in an MAU in order that the new record may be inserted in that MAU. Frequently, however, more than one MAU may have records which satisfy the mandatory clustering condition accompanying a record. So the user may also specify one or more optional clustering conditions, which are queries formed from clustering keywords, for the record to be inserted. With each of the optional clustering conditions is associated a weight. The insertion process which determines the MAU in which the record is to be placed is as follows: The set of MAUs containing records satisfying the mandatory clustering condition is first determined. For each of the MAUs in this set, a cluster weight is calculated by summing the weights associated with those optional clustering conditions that are satisfied by one or more records already existing in the MAU. The record to be inserted is then placed in the MAU whose cluster weight is the greatest.

E. DBC Commands

The front-end computer communicates with the DBC by issuing DBC commands. Two types of commands are recognized by the DBC: the access commands and the preparatory commands. Access commands are used to retrieve, insert, delete, and update DBC records in a file. Preparatory commnads are issued to manage file information and security specifications in preparation of subsequent access commands. Seventeen different commands have currently been provided.

A file may be opened for access by the open-database-file-for-access

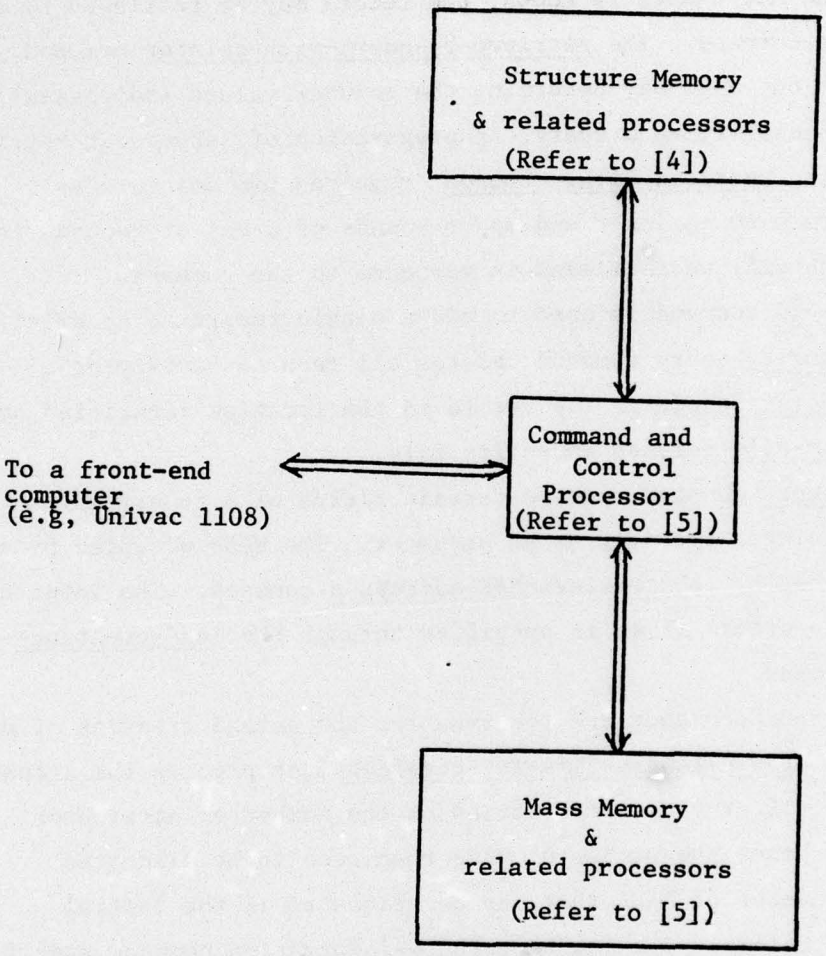


Figure 2.1. Basic architecture of the DBC

command and closed from further use by the close-database-file command.

Insertions, deletions and retrievals can be done on a file, once the file is opened for access. There are four commands to retrieve records from a file. In the retrieve-by-query command, a query composed of keyword predicates in disjunctive normal form is used to identify records desired by the user. If the MAU number is known, the record may be retrieved by a retrieve-by-pointer command. The retrieve-by-query-with-pointer command is provided so that the user may determine the pointer values (addresses) of a set of records satisfying a query, in preparation of subsequent retrieve-by-pointer commands. Retrieve-within-bounds requires two pointers as arguments, which are used as lower and upper bounds of a set of records in a file, all of which will be retrieved in response to the command.

The insert-record command is used to add a single record to an existing database. The delete-by-query command deletes all records satisfying a query. Delete-by-pointer deletes the record in the location identified by a pointer and delete-file deletes an entire file.

The replace-record command updates certain fields of a record,which is specified by a pointer provided as an argument. The MAUs occupied by a file may be determined by the retrieve-MAU-addresses command. The identity of the users who may create files is specified through the load-creation-capability-list command.

The remaining four commands are reserved for the actual creation of a database file. The open-database-file-for-creation must precede the actual creation of a file. It provides information on the number of attributes that the file is to have, the number of MAUs that need to be allocated initially, and the number of MAUs that may be allocated if the initial allocation is insufficient. The load-attribute-information command specifies the attributes for the file and the load-security-description provides information on the security descriptions for the file. Finally, the load-record command is used in order to create the file by loading the initial records.

In summary, each DBC record is a collection of attribute-value pairs. Some of these pairs are called keywords so that the DBC can associatively search MAUs for records containing any given keywords. Some of the keywords are designated Type-D. The DBC makes a directory entry in the structure memory(SM) for every Type-D keyword. Any Type-D keyword can also be designated as a clustering keyword or as a security keyword or both. The DBC automatically clusters records having identical clustering keywords. The security keywords are used for access control. The DBC provides a host

of update and retrieval commands to facilitate its use. It also has a hardware sorting capability in order to transmit records to the front-end computer in a sorted order.

3. THE DBTG DATA MODEL

The DBTG data model is of the network type. A complete language specification of data definition and manipulation facilities is presented in the April'71 report of the CODASYL Data Base Task Group [2]. In this section we shall extract the most important features of the network model that have been supported or defined in this language. We will not be concerned with the syntax of the language.

A DBTG database is defined in a schema by the database administrator. The schema consists of four sections, namely:

(1) schema entry
(2) area entry
(3) record entry
(4) set entry.

The schema entry merely provides a name for the schema itself. The area entry names the logical areas that together constitute the database. There is a record entry for each record type existing in the database and there is a set entry for each set. The terms record, area and set will be explained in the following sections.

3.1 The Record Type

The DBTG record is similar to a COBOL record. A record type (or record name) is defined as a collection of hierarchically related data item names or field names. The hierarchy of field names is defined by a template in the schema record entry. Any occurrence of the record type, or simply a record, will then have specific values for these data tiems. Thus a record type or record name is a generic name for all the record occurrences that have the same template or schema definition. As an example, consider the following record type:

```
RECORD NAME IS EMPLOYEE
 02   NAME
        03  LAST-NAME   PIC X(10)
        03  FIRST-NAME  PIC X(10)
 02   SPOUSE
        03  SPOUSE-NAME
            04  LAST-PART    PIC X(10)
            04  FIRST-NAME   PIC X(10)
        03  SPOUSE-AGE    PIC 99
 02   SALARY      PIC 9(5)
```

In this example, the hierarchy of data items is shown by the hierarchical tree of Figure 3.1. Normally, a record occurrence is stored in the database without the data item names. The value of a data item for the record occurrence can then be identified by its position within the record occurrence, since the values for the data items are stored in a specific order.

Any data item within a record type may be declared to be an array by using the OCCURS clause. Reference to any element in an array is accomplished through the use of subscripts.

3.2 The Set Construct

Relationships between records in the DBTG model are indicated through sets. A set (or set type) consists of a single record type called the owner record type and one or more other record types called the member record types. Record occurrences of the owner record type are termed owners and of the member record types members. Thus a set type asserts the existence of associations between records of heterogeneous types in the database. This allows the designer to interrelate diverse record types and to associate various entities in the database into a network-like model of real-world database management problems.

As in a record type, a set type also has occurrences. Each occurrence of a set must contain one occurrence of the owner record type and a number of occurrences of each of its member record types. It should be emphasized at this point that the owner record of a set is prohibited from being one of the member records of the same set. All the occurrences of a set are pairwise disjoint. In other words, a record occurrence cannot appear in two different occurrences of the same set. We will now take up three short examples to illustrate how hierarchical (one to many) and network (many to many) relationships as well as relationships between records of the same type are represented by set constructs.

A. Hierarchical Relationship

Consider an organization consisting of many divisions, each of which is composed of many departments, each of which in turn employs many personnel [6]. The relationships may be expressed by the data structure diagram depicted in Figure 3.2. The arrows are labelled by set names and are directed from an owner record type to a member record type.

Every occurrence of the set type DIV-DEPT is uniquely identified by an occurrence of the division record type and consists also of zero or more occurrences of the department record type. Similarly, every occurrence of

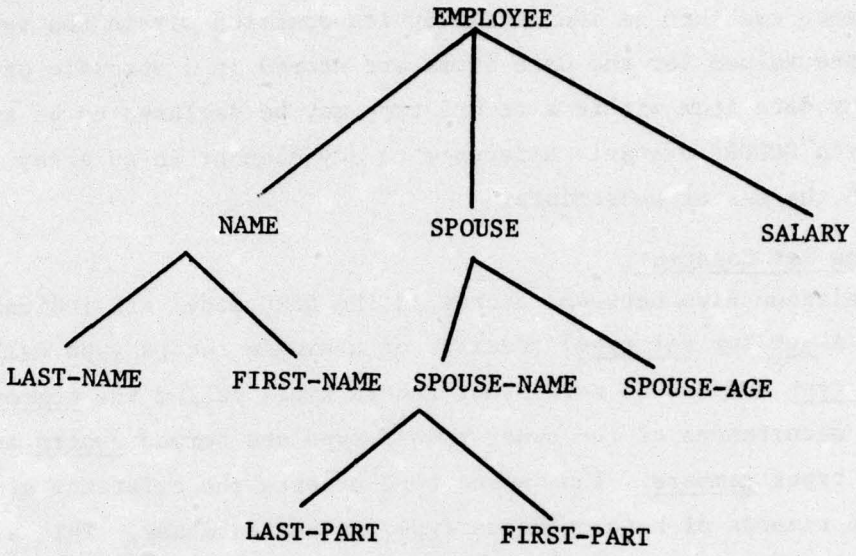


Figure 3.1. A hierarchical tree for a record

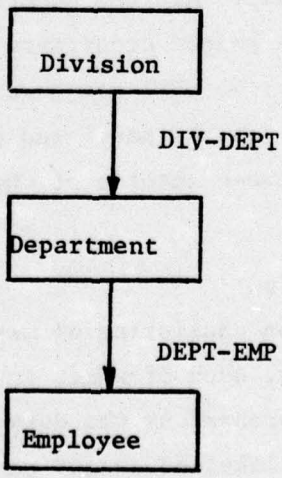


Figure 3.2. Data structure diagram for a hierarchical relationship

the set type DEPT-EMP is identified by an occurrence of the department record type and consists also of zero or more occurrences of the employee record type. In this representation no department can belong to more than one division and no employee can work in more than one department. The set occurrences are depicted in Figure 3.3.

B. Network Relationship

Very often two record types may have a many-to-many relationship to one another. Consider, for example [7], student and course record types. A student may be taking many courses and a course may have many students. To model this, it is not possible to construct the data structure diagram of Figure 3.4. This is because a course with more than one student would simultaneously be a member in two set occurrences of ENROLLED-IN, violating the rule of unique ownership. Similarly, when a student takes more than one course, the HAS-ENROLLED set condition is violated.

The usual way of representing many-to-many relationships is to define a third record type as shown in Figure 3.5. This new record type is used to relate the two other record types; it contains information that pertains specifically to both students and courses, e.g., grades.

C. Relationship Among Records of the Same Type

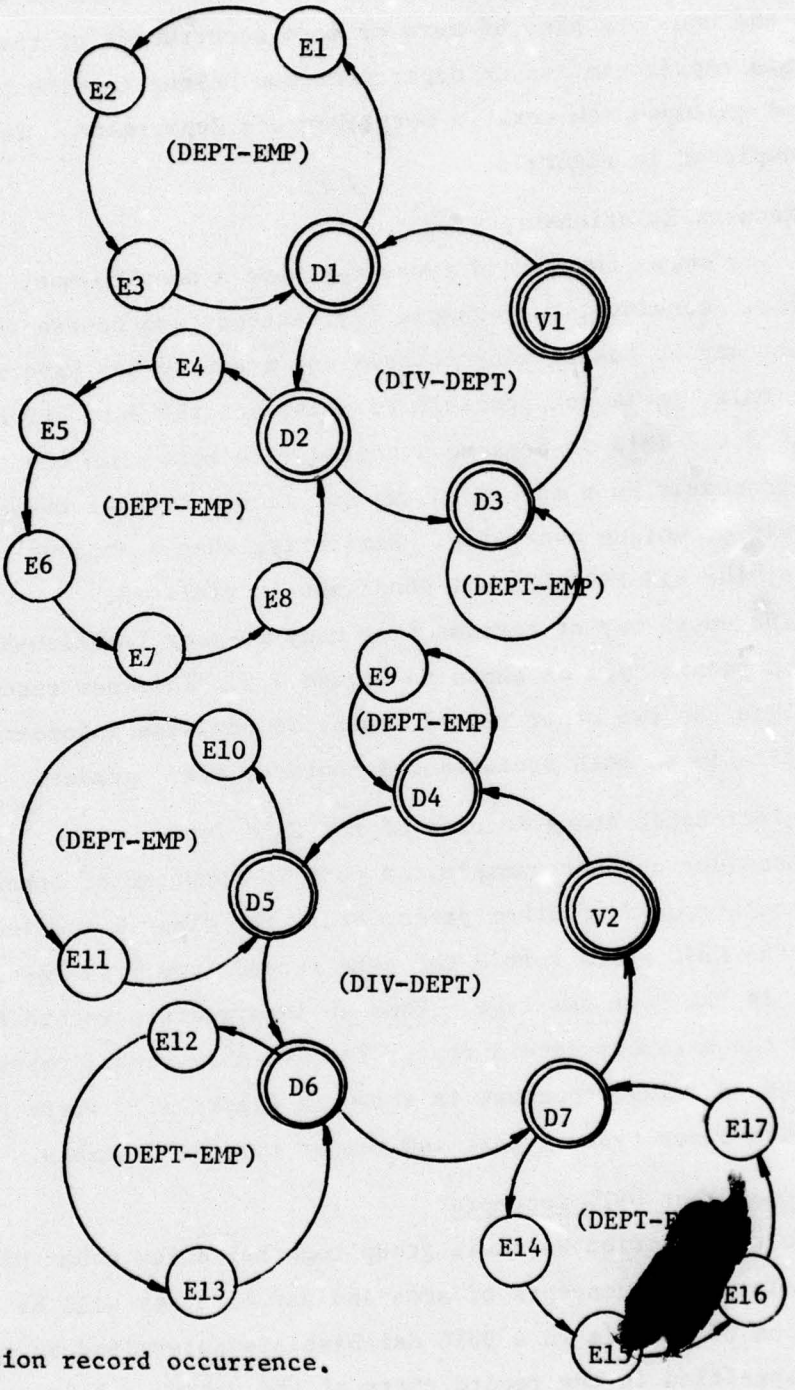
Consider another example. A part is composed of other parts, which in turn are composed of other parts, etc. The diagram in Figure 3.6 is illegal since the DBTG rules forbid the same record type from being both owner and member in the same set type. Thus an intermediate record type is introduced called the assembly record type. The assembly record represents an assembly of subparts. The structure is shown in Figure 3.7, where there are two sets each with owner type as part and member type as assembly.

3.3 Some Other DBTG Concepts

In this section we shall group together a few other DBTG concepts for discussion. The concepts of area and database key will be introduced. The locations of records in a DBTG database are determined by record location modes specified in the record entry of the schema. References to records are made with respect to only certain records called the current records. We shall also comment on the privacy features provided in the DBTG database model.

A. Areas

The database may be split into logical subdivisions called areas. The schema definition of a record may specify the areas in which occurrences of



: A division record occurrence.

: A department record occurrence. (Set-Name) : A set occurrence.

: An employee record occurrence.

Figure 3.3. A hierarchical relationship---two set occurrences of DIV-DEPT and seven set occurrences of DEPT-EMP

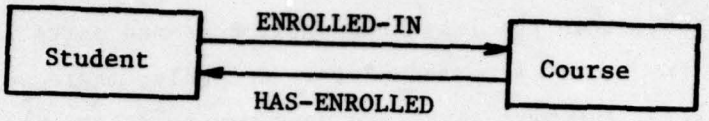


Figure 3.4. An illegal data structure diagram

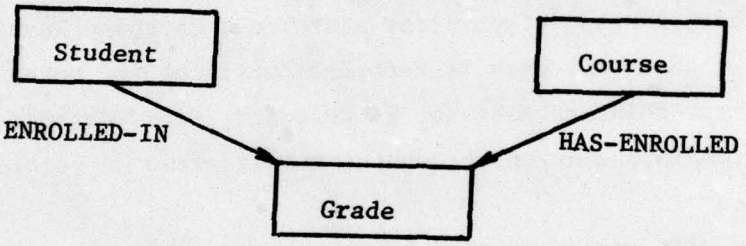


Figure 3.5. Data structure diagram for a many-to-many relationship

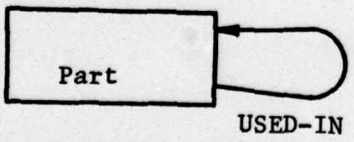


Figure 3.6. Illegal data structure for parts and assembly of parts

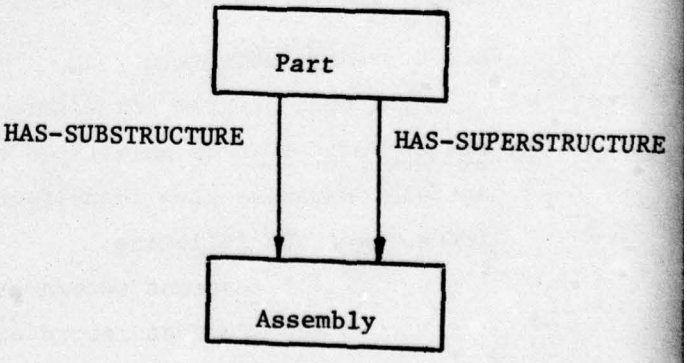


Figure 3.7. Data structure diagram for parts and assembly of parts

the record type may be placed. Each record occurrence, however, is placed in only one area. There may be several reasons for dividing a database into areas. Some parts of the database may be used only occasionally. If the database is divided into areas, then the less frequently accessed parts may be stored away from more frequently accessed parts. Secondly, there may be reasons of database security. The more sensitive data may be stored in areas with stringent protection. Physical clustering of records to facilitate certain processing modes (such as sequential or direct) may be another reason.

B. Database Keys

When a record occurrence is first stored in the database, the DBTG system assigns it a database key. A database key is a unique identifier of a record occurrence. Thus, instead of physical addresses, database keys may be used as pointers. In this way, even if record occurrences are moved around in the storage, their pointers need not be modified, since database keys are permanent and remain unchanged throughout the lifetime of record occurrences in the database.

In a given DBTG implementation, however, a database key does have some physical implications. For example, in UNIVAC's DMS 1100 [8,9], a database key is made up of an area name and an area key. The area key, in turn, is composed of a page number within the area and a record number within the page. In this way, a record occurrence with a particular database key is forced to reside in a specific page (a fixed-size block) in the database. However, it can be moved around within the page if a local page index is maintained in every page. If the page determined for a record occurrence is full, the record may be placed in an overflow page and linked to the original page.

C. Currency Indicators

For each program (also referred to as a run-unit) a table of currency status indicators is maintained by the DBTG system. These indicators are actually database keys identifying the most recently accessed record occurrence for each of the following:

(1) current record occurrence of the run-unit
(2) current record occurrence of each area
(3) current record occurrence of each set type
(4) current record occurrence of each record type
(5) current set occurrence (identified by the owner record occurrence) of each set type.

Most data manipulation statements refer to the current record occurrence of

the run-unit. Some others are with respect to the current record occurrence of an area, set or record type. Insertion of a member record occurrence into a set often requires the selection of a set occurrence, which may be the current set occurrence.

D. Record Location Modes

The location mode of a record type determines the DBTG strategy to be used for initial record placement in the database when a new record occurrence is being stored. The DBTG record schema allows for three different location modes for records. However, all occurrences of a given record type must have the same location mode.

The location mode of a record is said to be direct when the user is allowed the facility of specifying the database key for each record occurrence stored in the database. The record occurrence may then be placed in an area and location determined by the database key. It is the user's responsibility to remember the database keys of these record occurrences in order that they may be used for later reference.

If the location mode is defined to be calc, then the database key for a record occurrence is computed by a procedure that uses some combination of data items within the record as arguments. The data items are called calc keys. The calc procedure may be either implementor-defined or defined by the database administrator.

The location mode of a record may also be declared to be via a set in which the record type is a member record type. In this case, the DBTG system first selects an occurrence of that set based on the set occurrence selection criterion (to be discussed in Section 3.4). It then uses the set ordering policy (also discussed in Section 3.4) to determine the logical positions of the record within the set. It finally places the record occurrence in such a position that adjacent record occurrences within a set occurrence are physically "close" or clustered.

E. Privacy Features

The DBTG proposal allows for privacy through locks and keys. For every operation on each area, set type and record type, a privacy lock (or a procedure to determine such a lock) may be specified in the schema. A run-unit which requires a particular operation on an area, record or set, will have to provide a matching key before the required operation is carried out. The operation may be opening or closing an area; retrieving, deleting or creating a record occurrence of a particular record type belonging to a particular area;

inserting a record into a set; etc. The DBTG privacy mechanisms are not content-dependent in the sense that a record occurrence can not be locked out based on its contents. Either all records or no records of a particular type in a given area are allowed to be accessed.

3.4 *Set-Related Features*

In this section we shall explain certain important aspects of the DBTG data model that pertain to sets. In particular, we shall discuss membership class, implementation mode, set ordering, indexed sets, search keys and set occurrence selection.

A. The Set Membership Class

The membership of a record type within a set may be declared to be one of the following four classes: automatic mandatory, automatic optional, manual mandatory, and manual optional.

If the membership of a record in a set is automatic (whether it is mandatory or optional), then whenever an occurrence of a member record type is stored in the database, it is automatically made a member of an appropriate occurrence of the set determined by the set occurrence selection criterion for the set. The responsibility of inserting any new member record occurrence into a set is left to the run-unit if the membership is manual.

The membership of a record type in a set being mandatory implies that once the membership of a record occurrence in that set is established, the membership is permanent. Its set occurrence may be changed by an appropriate command, for example modify, but the record occurrence cannot be removed from the set without deleting it from the database. If the membership is optional then a record occurrence may be removed or reinserted into the set by the run-unit.

B. Set Implementation Modes

The manner in which a set is implemented is indicated through its mode. Two possible modes have been defined by the DBTG. If the mode is chain, then a chain of pointers is created which can be followed and which provides for serial access to all records in the set occurrence. The pointers are normally embedded in the records. If the mode is pointer array, then a member record in a set occurrence does not point to the next member record. Instead, the owner record contains a list of pointers (i.e., the database keys of the member records) which point directly to its member records.

C. Set Ordering

The ordering of the member records in a set occurrence is determined by the schema set-entry specification. The order in which the member records

are inserted in a set occurrence may be either in a sorted order or in an order dependent on the time sequence in which they are inserted and the position of the current record in the set. A member record may be inserted first (or last) meaning that its position in the set is next (or prior) to the owner record of the current set occurrence. If the ordering of a set is declared to be next (or prior), then any member record occurrence will be inserted in a position next (or prior) to the logical position of the current record of the set. On the other hand, if the set is declared to be sorted by some data items (e.g., database key) of the member records, then the logical position of a member record in a set occurrence will be such that all the member records in the set occurrence are sorted by the specified data items.

D. Indexed Sets and Search Keys

Any set declared to be sorted may also be declared to be indexed. This causes the system to build an index on the basis of the sort keys specified for each occurrence of that set.

An arbitrary number of search keys may also be specified for a set regardless of whether it is sorted or not. The arguments for such search keys must be data items included in the member records of the set. The declaration of a search key causes the system to develop some form of indexing for each occurrence of the corresponding set.

E. The Set Occurrence Selection

The system may be asked to select a set occurrence among all the occurrences of a set type. This is done by declaring a set occurrence selection clause for each member record type of the set entry in the schema. Automatic set occurrence selection is necessary when the system is required to find a set occurrence in which a member record occurrence is to be inserted or found. Therefore, there is a set occurrence selection clause for each member record type and one for the owner record type. For any particular member record type, its set occurrence clause will be used in the following circumstances to determine the appropriate occurrence of the set in which the member record occurrence is to be inserted or may be found:

(1) During the execution of a particular type of find statement (namely, find member-record-name via set-name), a set occurrence of the named set is determined using the set occurrence selection clause for the member record type. An appropriate member record occurrence is then found within that set occurrence.

(2) During the execution of a store statement, when the member record type to be stored is an automatic member of one or more sets.

(3) During the execution of a modify statement which changes the value of a data item specified in a set occurrence selection cluase.

A set occurrence selected may be the one containing the current record of the set. Another case may be where an owner record occurrence of the set is first selected using its database key or calc keys. This owner record then identifies the required set occurrence. There are many other methods of set occurrence selection. We shall reserve a more detailed discussion until Section 4.

3.5 Some Data Manipulation Facilities

The user writes his programs using a general-purpose language that hosts the DBTG data manipulation language (DML). DML facilitates operations on sets, performed usually through navigation of the sets. The starting operational point of most DML statements is the current record of the run-unit. Others can be based on current set occurrence or the current record occurrence of a set, area or record type. A find statement may be used in order to establish a record occurrence as the current record of the run-unit and also (optionally) as the current record of an area, record type or set. The delete statement may be used to delete the current record occurrence of the run-unit and to delete also the mandatory members of all sets in which the deleted record is an owner. The get statement retrieves the current record of the run-unit and places it in the user working area. A store statement is used in order to place a new record occurrence in the database. To manually insert record occurrences into sets, the insert statement is employed; and any optional member record occurrence may be removed from a set by using the remove statement. To modify the values of data items in a record occurrence, the modify statement is used, which may also change the membership of the record occurrence from one set occurrence to another (of the same set type) if the data items modified are those appearing in an appropriate set occurrence selection clause of the schema. The syntax of each statement, together with its detailed semantics may be found in [2].

For each record type defined in the schema, the system maintains a user working area(UWA) for that record type. Thus, at any instant, one occurrence of each record type may be stored in the UWA, for ready reference by the run-unit.

4. DBC REALIZATION OF A DBTG DATABASE

In representing a DBTG database on the DBC, the primary goal is to preserve the original information such that all operations performed on the DBTG database may still be performed on the DBC database with the same effect. To a user (or user program), the DBC implementation will be transparent. Aside from noticeable improvement in performance, the user will not be able to determine whether the implementation is on a conventional computer or on the DBC.

4.1 A Matter of Representation

A DBTG database is usually accompanied by a collection of indexes. An index is maintained for each search key declared in the schema. We, however, intend to represent the entire DBTG database without the use of such indexes. This will be possible due to the content-addressability of the DBC. Secondly, any record in the DBC must contain as keywords all information on which a user may choose to conduct a search. Thirdly, we would like to locate a record without navigating through a sequence of other records and this implies the elimination of pointers within records. Under the above guidelines, let us take up the problem of representing a DBTG database.

Since directories are to be minimized, most of the search-related information must be stored only as part of the DBC record. In addition, any other information related to a record will be preserved as attribute-value pairs. We may recall that the DBC can conduct an associative search based on queries that are composed of conjuncts of keyword predicates;and each keyword predicate consists of an attribute, a value and a relational operator. It is important, therefore, to represent as keywords of attribute-value pairs all information that can become part of a search argument. Any other information may be stored in the record body as non-keyword attribute-value pairs.

Since the content-addressability of the DBC is restricted to individual MAUs, the search space should be limited to as few MAUs as possible. For this reason, some of the keywords will be designated as type-D keywords while others will be treated as clustering keywords. A discussion on the proper choice of such keywords will be found in Section 4.3. Both type-D and clustering keywords require storage and processing in the structure memory (SM).

4.1.1 Representation of a Record

We now consider what information must be included in DBC records so that they can collectively represent every feature of the DBTG records. A record occurrence in a DBTG database belongs to a particular record type. It exists

in one and only one area. It has a unique database key. If it is a member of some set occurrences, the record occurrence must contain information that indicates the set occurrences to which it belongs as well as its logical position within these sets.

A. Representing the Record Type

The characteristics of all the record types are specified in the schema. Occurrences of these record types are stored in the database. Thus, every record occurrence belongs to a particular record type. We indicate this fact by including the following keyword as part of the DBC record representing the record occurrence:

<REC-TYPE, record-type>

where REC-TYPE is the attribute and record-type is the value; record-type is one of the record names defined in the schema and the one to which the particular record occurrence belongs.

B. Handling the Area

An area is a logical subdivision of the database. We have discussed in Section 3 some of the reasons a logical subdivision may be useful. The various areas are specified in the area section of the schema, for example,

AREA NAME IS MEDICAL-AREA

where MEDICAL-AREA is destined to be a logical area name.

In the record section of the schema, a record type may be specified to belong to one or more areas. However, when an occurrence of the named record type is physically placed in the DBTG database, it can only be placed in exactly one area. The necessary area name is provided by the run-unit through an area identifier. For example, consider the schema definition

```
RECORD NAME IS EMPLOYEE
    . . .
    WITHIN MEDICAL-AREA, EMP-AREA  AREA-ID IS ID-1
    . . .
```

Before storing a record occurrence of EMPLOYEE, the user can indicate the necessary area to be used by initializing ID-1 to either MEDICAL-AREA or EMP-AREA, but not both.

We represent the fact that a DBTG record occurrence belongs to a given area by including in the corresponding DBC record the keyword

<AREA, area-name>

where AREA is the attribute and the value, area-name, is the name of the area in which the record is to be placed.

An area may be declared to be temporary in which case it is not shared among concurrent run-units. Thus a temporary area is private to a run-unit. For this reason, we use a keyword

<AREA, run-unit-id.area-name>

where the area name is qualified by the run-unit id. After a run-unit terminates, all its temporary areas can be recovered by deleting all DBC records containing a keyword of the above type, i.e.,

<AREA, run-unit-id.area-name>

where area-name is one of the temporary areas used by the run-unit.

C. Database Keys and the Key Space

A database key is associated with every record occurrence to uniquely identify it from all other records in the database. Although it is supposed to be a logical identifier, its detailed structure in a DBTG implementation is usually not without some physical implications. It is necessary to determine a physical address of a record occurrence based on its database key. This determination is more easily done if at least a portion of the database key identifies a portion of the physical address.

In DMS 1100 [8,9], for example, a database key consists of an area code and an area key. Every area is assigned a unique code among all the areas. Each area is allocated a number of pages or fixed-size blocks in the secondary storage. The area key consists of a page number within all the pages allocated to an area, and a record number within the page. Thus, in DMS 1100, a database key consists of

(1) an area code

(2) a page number

(3) a record number.

The database key for a record occurrence may be generated by the system or it may be determined by the run-unit. The manner in which a database key is to be generated for occurrences of a specific record type is governed by the location mode of the record type. For example, the schema definition of a record type may look like

RECORD NAME IS R-1

LOCATION MODE IS DIRECT AREA-KEY, AREA-NAME

. . .

This specifies that before trying to store an occurrence of record type R-1, the run-unit will initialize the variables AREA-KEY and AREA-NAME, which together constitute a database key. In this case the run-unit directly

determines the database key.

There are two other record location modes. One is based on a calculation on certain data items of a record. For example,

```
RECORD NAME IS R-2
    LOCATION MODE IS CALC CALCSTATE IN NAMED-AREA
    USING STATE-NAME DUPLICATES ARE NOT ALLOWED
    . . .
02 STATE-NAME PICTURE IS X(12)
    . . .
```

In this case, it is specified that the procedure CALCSTATE must be used with the value of the data item STATE-NAME as argument to determine a page number and a calc-chain number within the area called NAMED-AREA. The DUPLICATES ARE NOT ALLOWED clause indicates that no two record occurrences may have the same calc keys, that is, they cannot have identical values for the data item STATE-NAME.

The other location mode is via a set. For example,

```
RECORD NAME IS EMPLOYEE-EDUCATION
    . . .
    LOCATION MODE IS VIA EMPLOYEE-HISTORY SET
    . . .
```

In this case, the database key (and thus the location) of an occurrence of EMPLOYEE-EDUCATION record is determined on the basis of its participation as a member in the set called EMPLOYEE-HISTORY. The area for the record occurrence is specified by the run-unit through an area identifier (unless there is only one possible area for the given record type). But the area key is determined by using the fact that the record occurrence may be located near its adjacent fellow-members in the EMPLOYEE-HISTORY set.

In the DBC, a database key need not have any physical implications. We shall only be concerned with the fact that in some cases the database key is created by the system and in other cases the run-unit is given control of determining a database key. We therefore create a key space of integers and divide it into two parts. A key space will be a set of integers from 1 to N where N is a very large number, usually a few orders of magnitude larger than the maximum number of records that can exist in the database at any given instant. A new record occurrence will be assigned the next unallocated number between 1 and N/2 as the database key, if the database key is to be generated by the system. If, on the other hand, a key is determined

by the run-unit, then the assigned database key is made N/2 larger than the user-specified key. This is shown in Figure 4.1.

It must be remembered that the database key is there in order that the system may refer to the various record occurrences. We may refer to a record occurrence not merely by the database key but also by other means. For example, when location mode of a record is direct, we may refer to a record occurrence by its database key as well as its record type. When location mode is calc, then the calc keys (data items in the calc clause) will be used, besides the record type. In this case the database key need not be used at all. Finally, when the location mode is via a set, then there will be other means to locate a record as discussed in Section 5. Once the database key has been determined in the manner described above, a keyword of the following form will be included in the DBC record:

<DBKEY, database-key>

where DBKEY is the attribute whose value is the database-key of the record.

The DBTG model allows for a special record type called SYSTEM, which can have only one occurrence. The database key of zero will be reserved for the DBC record which represents this record occurrence.

D. The Notion and Assignment of L-numbers

A conventional DBTG implementation generates database keys based on the location mode. A database key will then actually identify a physical address. This leads to a degree of data dependence which we intend to overcome in the DBC implementation. Instead of allowing a database key to represent a physical address, the DBC can maintain database keys in the structure memory (SM). Every database key <DBKEY, database key> could be declared a type-D keyword. Thus, given a database key, the DBC determines the MAU number of its corresponding record. However, an abnormally large amount of storage would be required since there will be a directory entry for every record in the database.

Let us take a closer look at the problem. The DBTG data manipulation features may require us to locate a record based on its database key if location mode is direct, based on certain data items if location mode is calc and based on its participation in a set occurrence if location mode is via a set. We will use this knowledge directly to determine a strategy to locate records.

Although not specified in the April '71 DBTG report, most implementations require that the number of pages required for every area and also the size of the pages in units of words or bytes be specified (see, for example, [8,9]). Since at a particular installation, the MAU size of the DBC will be known, it will be possible to determine the number of pages that may be fitted into

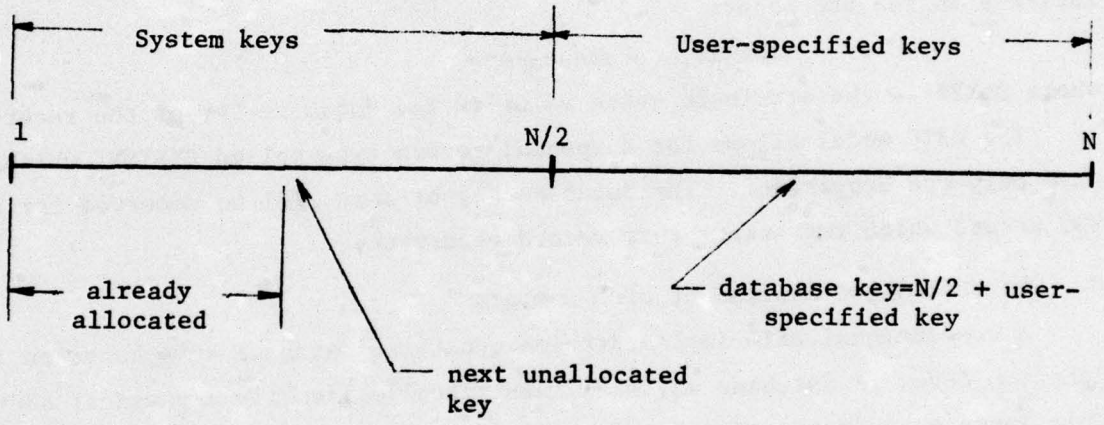


Figure 4.1. Determination of database key for the next record occurrence to be stored in the database

a single MAU. We can thereby determine the number of MAUs required for the entire database. A scan of the schema will also allow us to determine the number of record types in the database.

We will introduce at this point the concept of L-numbers. An L-number is a logical number assigned to every record occurrence in the database. It will not be used to identify a record but rather to aid the DBC in locating the MAU housing the record. We shall see in Section 4.3 that both record type and L-number will be used in locating records in the DBC. An L-number will actually act as a record type partition number. All record occurrences of a particular record type will be placed in disjoint partitions. Every such partition can be identified by a record type and an L-number.

The assignment of L-numbers is as follows. Let there be n MAUs and r record types in the database. Thus all occurrences of each record type can be accomodated in m (=n/r) MAUs on the average. We therefore need to assign m L-numbers uniformly among all the occurrences of each record type. We can thereby hope that all occurrences of a given record type and having a given L-number will fit into a single MAU. However, because of the variability of the number of occurrences per record type, we shall use mp L-numbers 1,2,...mp, where p>1. Possibly p=4 or 5, which is a design decision.

If the location mode is direct, then the database key of a record occurrence is hashed in order to determine its L-number from the range 1 through mp. If the location mode is calc, then the calc keys are hashed to an L-number. Finally, if the location mode of a record is via a set, then the appropriate set occurrence is first determined by using the set occurrence selection procedure for the given set. Once the set occurrence is determined, the L-number of the owner record of this set occurrence is known. We then assign this L-number to the record occurrence in consideration.

Once the L-number of a record occurrence to be stored has been assigned, the following keyword is included in that record occurrence:

<L-NUMBER, L-number>

where L-number is the one determined for the record occurrence. This keyword will be used both as a type-D keyword and a clustering keyword. Thus all records of a particular type with identical L-numbers will likely be stored in the same MAU. Besides, since the possible number of L-numbers is small (only mp of them), the size of the directory, and therefore the SM storage requirement, will also be small.

The same hashing procedure may be used for both the direct and calc location modes. The implementor may also choose to have different procedures. If, during the execution of a run-unit, a record occurrence is to be retrieved based on its location mode (direct or calc), then the L-number will first be calculated using the same hashing procedure that was used for storing the record occurrence. A DBC retrieval command will then be sent. The command will include the predicate (L-NUMBER=L-number) as part of the query. When location mode is via a set, the L-number only serves the purpose of clustering the records in a set occurrence; the location mode of such a record is never used for retrieval purposes.

E. Representing the Data Items of a Record

The structure of a record type defined in the schema is similar to a COBOL record with a hierarchical configuration of data items associated with appropriate level numbers.

```
RECORD NAME IS R
     ...
     02 A
        03 B
           04 C PIC 9(2)
           04 D PIC 9(2)
        03 E PIC X(5)
     02 C PIC X(5)
```

The terminal nodes of the hierarchy tree are called leaves, and for each leaf we need to store a keyword in every occurrence of this record type. Within a tree, if the names of all the terminal nodes are not unique, then they may be qualified by the names of nodes higher in the hierarchy such that the qualified names are all unique. For example, in the above record, the two terminal C-nodes are distinguished by qualified names B·C and R·C.

The keywords included for these data items will each have the qualified item name preceded by the word "RECORD." as attribute. The word "RECORD." is included only in order to indicate the fact that the attributes are associated with the data items in a record. Thus the keywords for the data items of this example are;

<RECORD.B'C, value>

<RECORD.D, value>

<RECORD.E, value>

<RECORD.R'C, value>

where value stands for the value of the appropriate data item of an occurrence.

Storing data items as keywords of attribute-value pairs increases the storage requirement since the data item names are stored (as attributes) in every record occurrence. However, in the DBC implementation, the data item names will be coded. Since there are usually only a few data item names within a record type, the codes will be small in size.

There is a tremendous advantage in storing data item values as keywords of attribute-value pairs. Since they are not type-D keywords, there is no storage overhead for directory maintenance. Yet, a random search can be efficiently conducted by the DBC for records containing arbitrarily specified data item values. To conduct a random search on arbitrary data items, a conventional DBTG system will require an index on every data item. In the absence of such an index an exhaustive search of the database will be necessary.

4.1.2. Representation of Set Membership

A set, as we have observed in Section 3, consists of one owner record type and one or more member record types. A set occurrence will be identified by an occurrence of its owner record and it may consist of an arbitrary number of member record occurrences. It is important to remember that all occurrences of a given set are pairwise disjoint implying that no tenant (owner or member) may exist in two occurrences of the same set type.

We shall now illustrate how the member records of a set are represented in the DBC and how their logical positions are indicated. A schema set entry consists of a set subentry and one or more member subentries. The set subentry names the set, defines its mode, its owner record type, and the logical ordering strategy for its members. Each member subentry names a member record type, its membership class (mandatory or optional, automatic or manual), the sort keys for ordering, any search key for which an index may be created and the set occurrence selection criterion.

A member record occurrence belongs to a set occurrence identified by an owner record occurrence. Let us assume that a record occurrence r is a member

of the set called set-type and that the corresponding set occurrence is identified by owner-database key, which is the database key of the owner record occurrence. We then include in r the keyword

<SET.set-type, owner-database key>

in order to identify the set occurrence in which it is a member. For each set in which r is a member, a keyword of this form will be included in r.

Although the database key of the owner record occurrence uniquely identifies a set occurrence, it is not enough to store only that in a member record occurrence. This is due to performance reasons. Given a member record occurrence, we shall often be required to locate its owner in a given set, and this cannot always be done from a knowledge of the database key alone. We also need to know the L-number (record type partition number) of the owner record occurrence. Assume, once again, that a record occurrence r is a member of the set called set-type and that the owner of the corresponding set occurrence has an L-number termed owner-L-number. We then include in r the following non-keyword attribute-value pair:

<OWNER-L-NUMBER.set-type, owner-L-number> .

A record type may also be declared to be the owner record type of an arbitrary number of sets. For each set of which a record occurrence r is an owner, we include in r the keyword

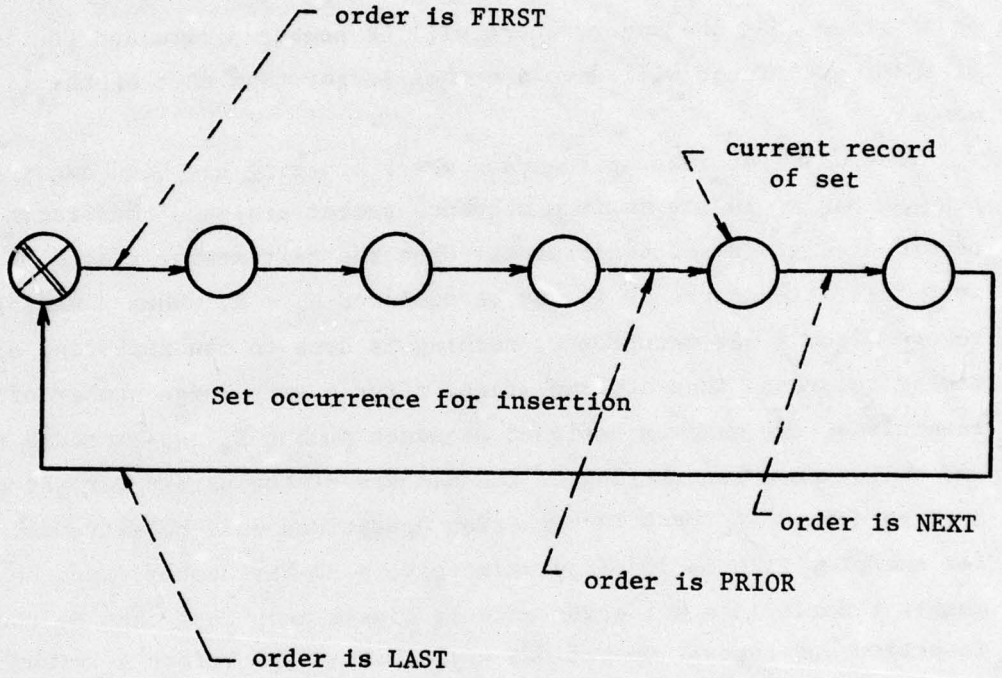
<SET.set-type, OWNER>

where set-type is the name of the set.

4.1.3 Preserving Set Ordering

It is not enough to show only the presence of a member in a set. In the DBTG model, the logical position of a member record within a set occurrence is also considered to be of significance. The member records may be ordered in two different ways. The ordering may be based on the time of insertion of a new entry into a set and also on the current record of the set. In the second method, the ordering is based on certain sort keys of the member records. We shall consider them in turn.

In the first type of ordering, the order of insertion may be declared in the schema to be first, last, next or prior. The insertion point of a member record occurrence in each of these four different cases is shown in Figure 4.2. It should be noted that only one of the four cases can be specified for insertion order. When the order is first, any new member record will be inserted next to the owner of the appropriate set occurrence. When the order is last, the new member record is inserted prior to the owner record occurrence. If the



: owner record occurrence

: member record occurrence

Figure 4.2. Insertion points to maintain set ordering

order is next (or prior), then the insertion point is next to (or prior to) the current record of the set.

A. Cases where the Set Order is First or Last

The ordering information may be maintained by assigning a sequence number to the member records of a set occurrence. The owner record is numbered zero, the first member is numbered 1, the second member is numbered 2, etc. This may be done easily when a set is first created. At any later stage, some deletions or removals from the set may change the numbering but it will always be the case that the owner record will be numbered zero and the i-th member of a set occurrence will have a number larger than that of the (i-1)-th member.

Let us first take up the case where ordering has been declared to be last. Assume that N_s is the maximum sequence number assigned thus far to any member record of a given set occurrence. When the next member record is inserted into this set occurrence it may be numbered $N_s + 1$. When a member record is removed from a set occurrence, nothing is done to the numbering of other member records. Thus at some stage, after a very large number of removals and insertions, the maximum assigned sequence number N_s may exceed a fixed number N, and only then a renumbering of the members of the particular set occurrence need be done. But these re-numbering operations will be extremely infrequent; for example, if N is 10^9 (approximately, a 32-bit number) and the number of member records in a set occurrence is always much less than N, then about 10^9 insertion and removal operations will be required before a re-numbering will be required.

A similar, but not identical, strategy will work in case the ordering is declared to be first. To insure that at any instant the owner of a set occurrence is numbered zero and the i-th member is assigned a sequence number larger than the (i-1)-th member, we may do the numbering in reverse.

Once the sequence number of a record (owner or member) within a set occurrence s is decided, it is stored as a keyword of the record in the form:

<SETPOSITION.set-name, sequence-number>

where set-name is the set to which the set occurrence s belongs.

Global information on a set occurrence, such as the current number of members and the maximum (minimum) assigned sequence number when ordering is last (first), will be stored in the owner record of the set occurrence. Thus, the owner record occurrence R of an occurrence of the set named set-name will contain the attribute-value pairs

<SET-MEMBER.set-name, p>

and <SET-SEQNUMBER.set-name, N_s>

where p is the current number of members in the occurrence of set-name, identified by the owner record occurrence R, and N_s is the maximum (minimum) assigned sequence number for the same set occurrence. These two attribute-value pairs need only be stored as a part of the record body and not as keywords, since they will never be used in any content-addressing command.

It is clear from the preceding discussion that in the DBC the insertion of a record into a set occurrence does not require any reference to other member records when set order is first or last. Only the owner record occurrence will have to be referenced in order to modify the current number of members, p, and the maximum (minimum) assigned sequence number, N_s. On the other hand, in a conventional DBTG implementation with pointers, it will be necessary to modify the link or pointer fields of one or two other records (next and prior to the inserted record) depending on whether the list of member records is singly or doubly linked.

B. Cases where the Set Order is Next or Prior

When the set order is next or prior, a numbering scheme for the DBC implementation is more involved. The numbering process will be discussed in two parts:

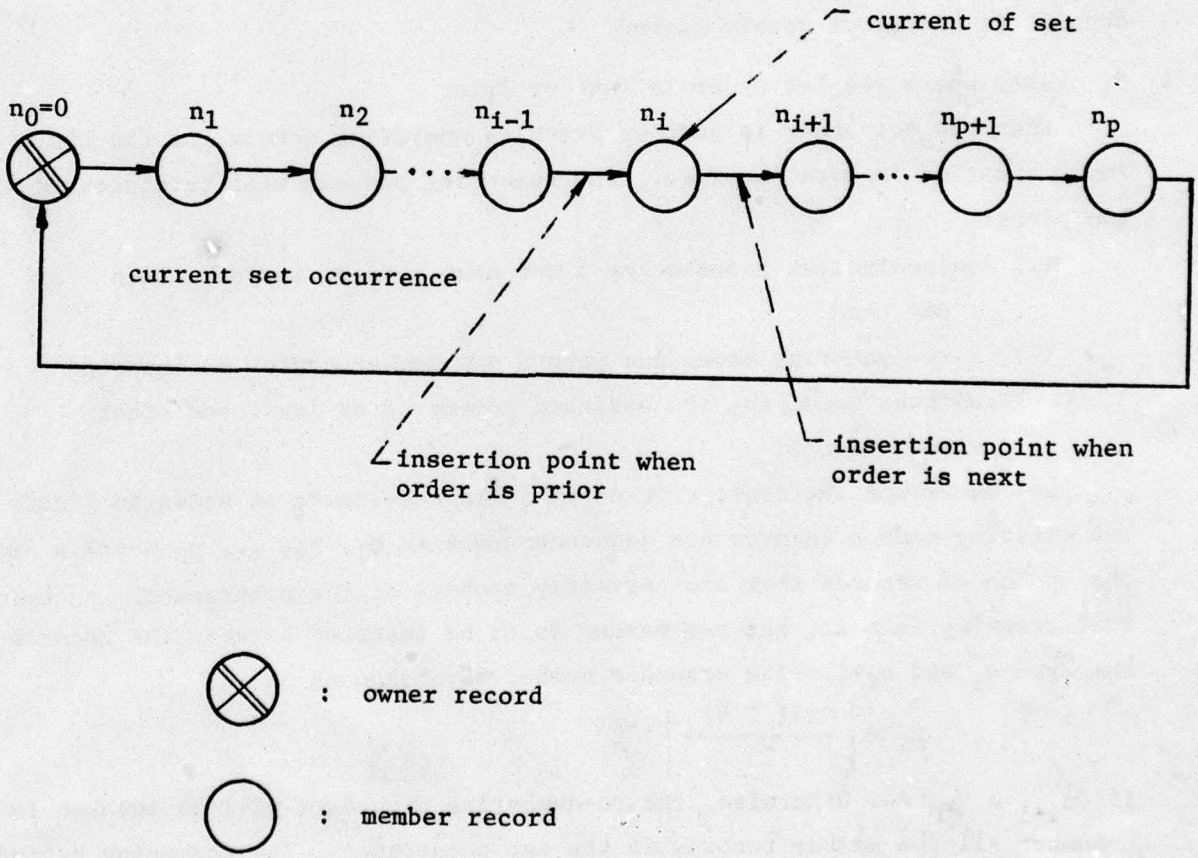
(1) an assignment process for a new member to be inserted into a set, and

(2) a re-numbering procedure when a new member cannot be inserted without modifying the assigned number of at least one other member record.

Let us assume the configuration of a set occurrence as shown in Figure 4.3. The existing member records are sequence numbered $n_1, n_2, \ldots, n_p$ where p is the number of records that are currently members of the occurrence. Assuming that ordering is next, the new member is to be inserted between the records numbered n_i and n_i+1. Its sequence number is chosen as

$$n_i + \left\lfloor \frac{n_{i+1} - n_i}{2} \right\rfloor$$

if $(n_{i+1} - n_i) > 1$. Otherwise, the re-numbering procedure will be invoked to re-number all the member records in the set occurrence. The reasoning behind the above scheme is as follows: whenever a number is to be selected between two numbers n_i and n_{i+1}, it is advantageous to select the new number halfway between the two. Assuming that at a later stage, a number may have to be selected with equal probability between the two halves thus created, there is a good chance that the same process may be repeated without resorting to



: owner record

: member record

The records in the set are labelled with their sequence numbers indicating their positions in the set occurrence,

$$n_0 = 0 < n_1 < n_2 < \ldots < n_p.$$

Figure 4.3. Sequence number generation of a new member when ordering is next or prior

re-numbering.

A new record is to be inserted as the last record of a set occurrence when the current record of the set is presently the last member in its set occurrence. In such a situation, if n_p is the sequence number of the last member in the set occurrence, then the new member is assigned a sequence number

$$n_p + \left\lfloor \frac{N-n_p}{2} \right\rfloor$$

if $(N-n_p)>1$, where N is a fixed, very large number. In case $N = n_p+1$, then a re-numbering will be required. Notice that the special case just discussed also takes care of the situation resulting from an empty set (that is, the set occurrence consists of the owner record alone). When the set occurrence is empty, then p=0 and $n_p = n_0 = 0$ and the new member record will be assigned a sequence number $\frac{N}{2}$.

In the event of a "collision" (i.e., when a sequence number is to be generated between n_i and n_{i+1}, but $(n_{i+1} - n_i) = 1$), then re-numbering is required for all the members in the set occurrence. This is done as follows. Let p be the number of members currently existing in the set occurrence (p is a count of the member records alone, the owner is not counted in p). Form a real number b as

$$b = \frac{N}{p+1}$$

The owner record is then given a sequence number $n_0= 0$, the first member is assigned $n_1 = \lfloor n_0 + b \rfloor$, etc. and the last member is assigned a sequence number $n_p = \lfloor n_0 + pb \rfloor$. After re-numbering, the new member may be assigned a number by the assignment process previously described. The re-numbering scheme ensures that there is an equal number of unassigned sequence numbers between any two assigned sequence numbers. The case of prior ordering can be handled in a similar fashion.

Once again, the keyword needed for the sequence number in every tenant (owner or member) of an occurrence of the set named set-name is

<SET POSITION.set-name, sequence-number>.

The owner record also keeps track of the number of member records p, by means of an extra attribute-value pair (which is only a part of the record body and is not a keyword)

<SET-MEMBER.set-name, p>

C. Ordering by Sort Keys

The members in a set may also be ordered by some sort keys. Special attribute-value pairs to indicate the ordering are only needed when duplicates

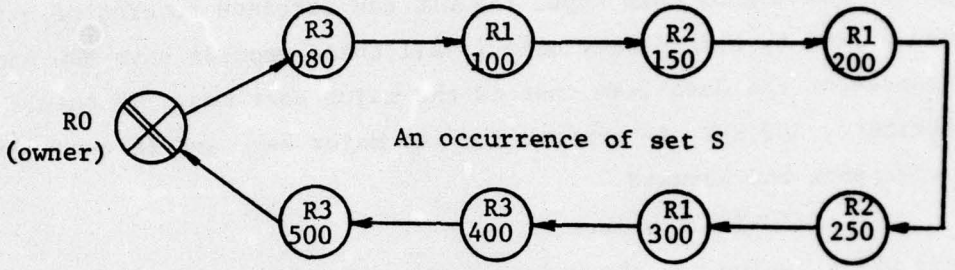
are allowed (i.e., when two or more members in a set occurrence are allowed to have identical sort keys). In this case, ordering information will be lost in the DBC representation unless a special keyword is inserted in those member record occurrences. There are a number of ways in which ordering may be established through sort keys. Let us consider each in turn. In each of these cases, we will take as example a set type S declared to have record type RO as owner and record types R1, R2 and R3 as members. Furthermore, we shall concentrate on a single set occurrence of S in which there are the following member record occurrences:

Type R1, DBkey 100, sort key 12
Type R1, DBkey 200, sort key 9
Type R1, DBkey 300, sort key 10
Type R2, DBkey 250, sort key 24
Type R2, DBkey 150, sort key 28
Type R3, DBkey 400, sort key 15
Type R3, DBkey 080, sort key 17
Type R3, DBkey 500, sort key 30

(1) Ordering on Database Keys - The members are ordered by their database keys, independent of their record types. Thus they are ordered as shown in Figure 4.4. Since database keys are unique, no special keywords are needed in the record occurrences to represent the ordering.

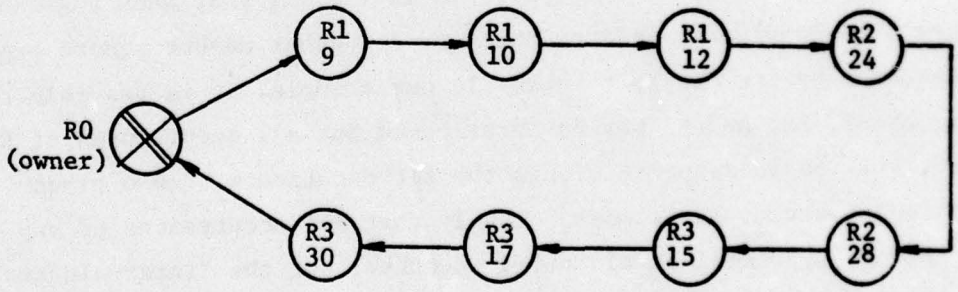
(2) Ordering on Record Types as well as Sort Keys - The major sort key for member records may be the names of their record type (or their record codes if every record type is assigned a unique integer as a code). The minor sort keys will be the sort keys of the individual record types as indicated in the schema. Figure 4.5 illustrates the configuration of our set occurrence with respect to such an ordering. As before, no special keywords are necessary to retain ordering information since such information is already stored in every member record occurrence in the form of its record type and sort keys. The special case arising due to duplicate sort keys will be considered later.

(3) Ordering on Sort Keys Irrespective of Record Types - This type of ordering is relevant if a set is composed of more than one member record type. Ordering on sort keys irrespective of record types means that the member records in a set occurrence are to be maintained in a single sequence regardless of the number of different member record types specified for the set. The corresponding sort keys for each member record type must, therefore, have identical data characteristics and must also match in terms of whether they are ascending or descending keys. For example, if RA and RB are two different



(member records are labelled with their types and database keys)

Figure 4.4. Members ordered by database keys



(member records are labelled with their types and sort keys)

Figure 4.5. Members ordered by record type and sort keys

member record types in a set, if RA is to be sorted by its data items A1 (major key) and A2 (minor key) and if RB is to be sorted by its data items B1 and B2, then A1 must have the same data characteristics (e.g., the same PICTURE clause) as B1, and A2 must have the same data characteristics as B2. In our example, assuming that R1, R2 and R3 have sort keys with identical data characteristics, our set occurrence will be as depicted in Figure 4.6.

With an ordering of this type, the DBC can retrieve records of a set occurrence sorted by a major key only if all these records have the same keyword to represent the data item that is the major sort key. To ensure this, a new attribute, SORTKEY, is created for the major key, and in each member record occurrence the keyword

<SORTKEY, value>

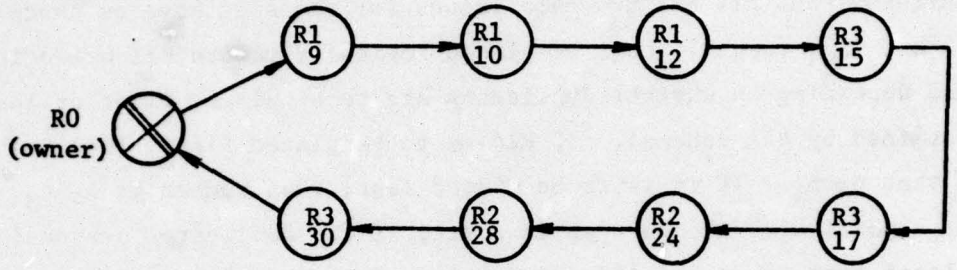
is stored, where 'value' is the value of the major key. For example, if record type RA is to be sorted in set S by data items A1 and A2, where A1 is the major key, then every occurrence of RA will have the following two keywords,

<RECORD.A1, value-of-A1> and <SORTKEY, value-of-A1>.

Since the DBC cannot sort by more than one keyword, it is unnecessary to store similar keywords for the minor keys. Sorting on the minor keys will be done in the interface buffer.

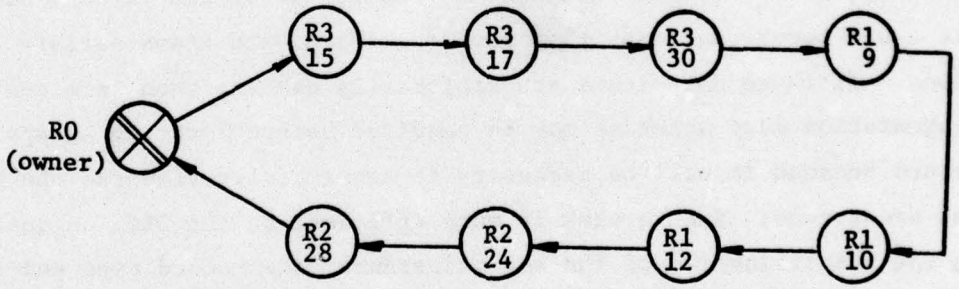
(4) Ordering on Sort Keys but Only Within Record Types - Within a set occurrence, instances of a member record type may be sorted without regard to the order of other record types in the set. This does not mean that there is an implied major sort by record type. It only means that when a given type of record is considered independently of any other member record type, it is in sequence by its own sort keys. In our example, it is possible that occurrences of R1, R2, and R3 may be interleaved but all occurrences of R1, for example, will be in sequence within the set occurrence. In a given implementation, however, it is most likely that all occurrences of any particular record type will be clustered together, but the "inter-cluster" sequence will not depend on the record type but may depend on the chronological order in which the first occurrence of each record type has been inserted into the set occurrence. Thus, a possible configuration of our set occurrence is given in Figure 4.7. Once again, no special keywords are needed in the member record occurrences unless duplicate sort keys of any record type are allowed. The special case where duplicates are allowed will be considered later.

(5) Duplicate Sort Keys - If two or more occurrences of a record type R in a set occurrence are allowed to have identical values for their sort keys, then the DBC has no way to know the ordering among these record occurrences.



(Member records are labelled by their record types and sort keys)

Figure 4.6. Members ordered by sort keys irrespective of record types



(Member records are labelled with their types and sort keys)

Figure 4.7. Members, of each record type, ordered by sort keys private to the record type

Hence, some extra information needs to be stored in the form of attribute-value pairs. For example, if three record occurrences, R11, R12 and R13, have identical sort key values, then they may be numbered n_1, n_2, and n_3, respectively, to indicate their positions with respect to one another. If R11 precedes R12 and R12 precedes R13 in the set order, then $n_1<n_2<n_3$.

Now, when another record occurrence R14 is to be inserted into this set occurrence, and R14 has the same values for the sort keys as those of R11, R12 and R13, then R14 must be placed logically before R11 or logically after R13 depending on whether duplicates are to be placed first or last (as determined by the schema). If R14 is to be placed first, then number it n_4 such that $n_4<n_1$. If it is to be placed last, then number it $n_4>n_3$.

For a particular record type in a set, if the duplicates are declared to be placed last, then all its occurrences with no duplicates are numbered zero. All subsequent duplicates will be numbered 1,2,3, etc. On the other hand, if duplicates are to be placed first, then all member record occurrences with no duplicates are numbered N, where N is a very large number. All subsequent duplicates will be number N-1, N-2, etc.

Thus, when duplicates are allowed in a set for a given record type R, all occurrences of R must have a numbering keyword

<SETPOSITION.set-name, sequence number>

where sequence number identifies the logical position of the record occurrence among its duplicates in the set occurrence.

(6) Search Keys - Sometimes a user may decide to search for one or more records in a set occurrence such that some specified data items satisfy certain given values. If these data items are arbitrarily chosen, then in a conventional DBTG implementation many accesses may be required before locating an appropriate record because it will be necessary to sequentially traverse the members of the set occurrence. Such a task is more efficient on the DBC. A query involving the identification of the set occurrence, the record type and the data items will force a search of a very few MAUs (usually only one).

The DBTG model allows indexes for data items declared as search keys in order to improve performance. Otherwise, there is no alternative to a sequential search. These indexes of course take up much primary memory. The DBC implementation, on the other hand, work out much better in these situations.

4.2 Summary of Keyword Assignment

In summary, each DBC record occurrence is made up of the attribute-value pairs described below. Each pair is a keyword unless explicitly stated otherwise.

(1) Type of the record occurrence :

<REC-TYPE, record-type> .

(2) Area in which it is stored :

<AREA, area-name> .

(3) Database key :

<DBKEY, database-key> .

(4) L-number of the record occurrence :

<L-NUMBER, L-number> .

(5) For each data item :

<RECORD.data-item, value> .

(6) For every set of which the record occurrence is an owner,

<SET.set-type, OWNER>

and a non-keyword attribute-value pair

<SET-MEMBER.set-type, current-number-of-members>.

(7) For every set of which the record occurrence is a member,

<SET.set-type, owner-db-key>

and <OWNER-L-NUMBER.set-type, owner-L-number> .

(8) If the chronological ordering of the member records in a set is required, then each member record occurrence will have the keyword

<SETPOSITION.set-type, sequence-number>

and the owner record occurrence will have the non-keyword pair

<SET-SEQNUMBER.set-type, maxm-or-minm-seq-number>.

4.3 Type-D Keywords and Clustering

The choice of type-D keywords is always based on the type of keywords that appear in a query. Every predicate conjunct in a query must have at least one predicate that consists of a type-D keyword. If this condition is not satisfied, then the query can be answered only by searching every MAU in the database. With the requirements of data manipulation in mind, we have decided on type-D keywords as all those that have one of the following attributes :

(1) REC-TYPE

(2) AREA

(3) L-NUMBER .

4.3.1 Clustering Methods

Some type-D keywords will also be made clustering keywords. We will primarily be interested in clustering set occurrences, since set traversal is the most important operation done on a network database.

A. Clustering Method I

We may cluster by L-numbers, because all members of a set occurrence have the same L-number if the location mode of the member records have been declared to be via that set. Thus an entire set occurrence will be accommodated in as few MAUs as possible.

B. Clustering Method II

A second clustering method is to cluster primarily by record type and secondarily by L-number. Thus all occurrences of the same record type will be placed in an few MAUs as possible. For example, if a record type R has 10,000 occurrences and each MAU can accommodate 2,000 of them, then it is conceivable that only 5 MAUs will contain all the occurrences of R. Clustering secondly by L-number will normally ensure that all occurrences of R that have the same L-number will be placed in the same MAU.

C. Choice of a Clustering Method

The first method is useful when many record types are located via a single set S. In that case, an occurrence of S is likely to be placed in a single MAU because all the members of that set occurrence have the same L-number. The second method will place an occurrence of S in possibly n MAUs if there are n different member record types.

On the other hand, if a record type R has a location mode direct or calc and is declared to be a member of a set S, then the second method is far better. In the first method, a set occurrence of S is likely to be scattered over many MAUs, a number not much smaller than the number of records in that set occurrence. In the second method, a set occurrence of S will spread over approximately m MAUs, where m is the number of MAUs required to contain all the occurrences of record type R, and this number is likely to be much smaller than the number of member records in a set occurrence. In our interface, therefore, we shall use the second clustering method. Incidentally, conventional implementations do not cluster by record type, because the database keys are user-specifiable in some cases, for example, when the location mode is direct. Thus, traversing a set S, whose members do not have a location mode via S, will require many more accesses in a conventional network database system than in the DBC.

4.3.2. Directory Memory Requirement

Directory entries are stored in the DBC structure memory for every type-D keyword. Our choice of type-D keywords is directed towards efficient processing of data manipulation operations and also towards minimizing the directory

memory requirement. What follows now is a somewhat gross analysis to substantiate the claim that directory memory requirement on the DBC is indeed very small.

Let the database consist of r record types, a areas and n MAUs. Let m = n/r. We shall then use mp L-numbers where p is a small number greater than 1. Let us further assume, for simplicity, that each type-D keyword requires four bytes of storage and each MAU number can be represented in two bytes.

Each directory entry will consist of a type-D keyword and one or more MAU numbers. Since we cluster records by their types, all records of a given type will be accommodated in n/r MAUs on the average. Since we cluster secondarily by L-numbers, all records with a given L-number will be spread over r MAUs (because all record occurrences of the same type and same L-number will possibly be clustered within a single MAU). If record types do not, in general, span more than a single area, then we may expect r/a record types to be assigned to each area. Therefore, each area will be spread over $(n/r)\cdot(r/a) = n/a$ MAUs. Thus, under the above assumption, records are automatically clustered by area as well.

We tabulate below the memory requirement for storing the directories in the structure memory.

Type of keyword	No. of such keywords in the directory	No. of MAU references per keyword	Total directory memory requirement (in bytes)
REC-TYPE	r	n/r	r(4 + 2n/r)
AREA	a	n/a	a(4 + 2n/a)
L-NUMBER	np/r	r	(np/r)(4 + 2r)

Thus, the total directory memory requirement is

$$(4r + 2n) + (4a + 2n) + (4np/r + 2np)$$

$$= 2(2a + 2r + 2n + 2np/r + np) \text{ bytes.}$$

As an example, if the database consists of 10,000 MAUs, 10 areas and 100 record types and if p is chosen to be 5, then the directory memory requirement is approximately 143,000 bytes. This is an extremely small fraction of the database size. In fact, if the MAU size is 10^6 bytes, then the directory memory size is less than 0.01 percent of the size of the database.

4.4 Privacy

The DBTG model protects the privacy of portions of the database through

a mechanism of privacy locks and keys. There are various functions that may be performed in the schema, on records as a whole and on the data items in a record. These functions may be assigned privacy locks in their schema definition. Matching privacy keys must be provided by a run-unit in order to perform these operations. However, the specifications of privacy locks are such that an operation may be performed on all the record occurrences or on no such record occurrences in an area (or of a particular record type). That is, the actual values of the data items in a record occurrence have nothing to do with its privacy. Therefore, we need not have any special representations of the record occurrences in order to ensure privacy. All privacy checks can be done easily in the interface. However, if the DBTG model is enhanced to provide for privacy features based on record content, then the very powerful content-addressing property of the DBC may be utilized to satisfy such privacy requirements. Thus, the DBC can conveniently support far more powerful privacy features in terms of data contents than that provided in the DBTG model.

5. THE TRANSLATION PROCESS

The DBTG data model provides the user with a moderately large number of operations. In this section, we shall illustrate the design of an interface to support these operations. The essential properties of the data manipulation operations will be extracted and simulated by an interface, called the DBTG interface (DBTGI) or interface module. The user programs issue the DBTG DML commands which are routed to the DBTGI. The interface, in turn, actually processes the DML commands by translating them into equivalent DBC commands and by sending the translated commands to the DBC.

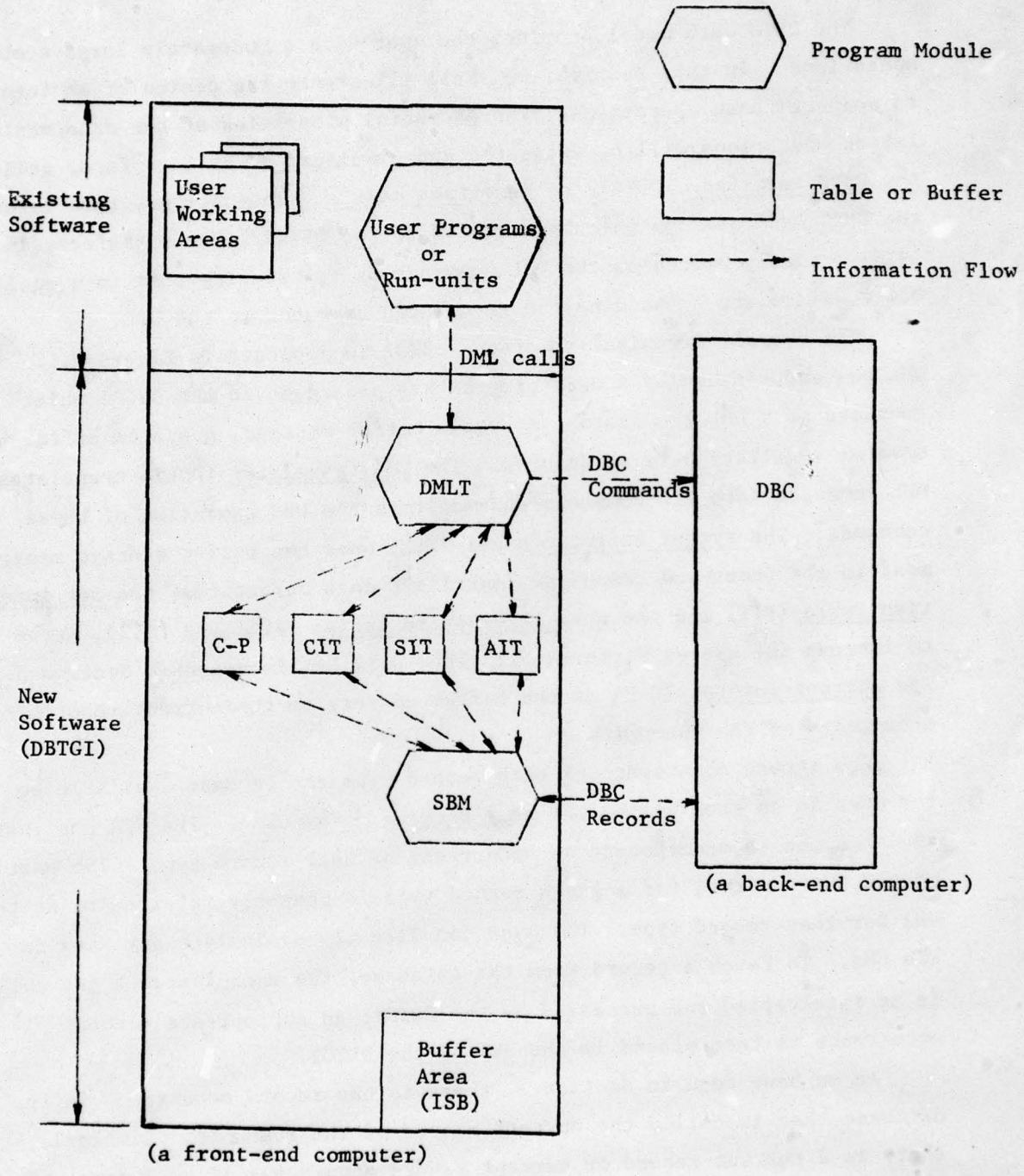
The overall organization of the DBTGI is depicted in Figure 5.1. The DML commands issued by a user program are passed on to the DBTGI which consists of a DML translator, a system buffer manager, a system buffer and several auxiliary data structures. The DML translator (DMLT) translates DML commands into DBC commands and monitors the DBC execution of these commands. The system buffer manager (SBM) does the buffer storage management in the front-end computer. Auxiliary data structures, the set information table (SIT) and the area information tables (AIT1 and AIT2), serve to improve the system performance. They will be discussed in Section 5.3. The current-pointer (C-P) is the buffer address to the current record occurrence of the run-unit.

One record occurrence of each record type can be made available to the user in an area called the user working area (UWA). The UWA has just enough space to accommodate an occurrence of each record type. The portion of the UWA reserved for a given record type is commonly referred to as the UWA for that record type. The user can directly manipulate any data in his UWA. To fetch a record from the database, the user issues a get call. It is intercepted and processed by the DBTGI; an appropriate record occurrence is then placed in the UWA by the DBTGI.

As we have seen in Section 3, there is one record occurrence in the database that is called the current record of the run-unit. Similarly, there is a current record or current record occurrence of each area, of each record type and of each set type. In addition, every set type also has a current set occurrence that is identified by an occurrence of the owner record. The current records are established by the run-unit using DML find statements. The DBTGI stores currency information in the currency indicator table (CIT). The information maintained in the CIT consists of:

(1) For each area

a) area name



DBTGI: DBTG Interface Module
SBM: System Buffer Manager
DBC: The Database Computer
C-P: Current-pointer (pointer to the current record of the run-unit)
DMLT: DML Translator
CIT: Currency Indicator Table
SIT: Set Information Table
AIT: Area Information Tables
ISB: Interface System Buffer

Figure 5.1. The interface, DBTGI

b) record type of the current record of the area
c) L-number of the current record of the area
d) database key of the current record of the area.

(2) For the run-unit:
a) record type of the current record of the run-unit
b) L-number of the current record of the run-unit
c) database key of the current record of the run-unit .

(3) For each record type:
a) record type
b) L-number of its current occurrence
c) database key of its current occurrence.

(4) For each set type:
a) set type
b) information about current set occurrence
i) owner record type
ii) L-number of the owner of the current set occurrence
iii) database key of the owner of the current set occurrence
c) information about the current record of the set
i) whether current record is a member or the owner
ii) record type of current record
iii) L-number of current record
iv) database key of current record
v) position information about current record
- either the sequence number in the set occurrence or the values of the sort keys.

5.1 Set Occurrence Selection

Whenever an occurrence of a set has to be selected automatically for the user program, the selection of a set occurrence with respect to a given member record type is governed by a set occurrence selection clause in the corresponding member subentry of the schema set entry. For example, suppose an occurrence r of an automatic member record type R of set S is to be stored in the database. The operation requires not only that the record be stored in the database but also that it be inserted into an occurrence s of set S. To select s, the procedure specified by the set occurrence selection clause for S with respect to R must be activated. Once s is selected, the record occurrence r is then inserted into s, in a position governed by the ordering specifications for S.

There are a number of ways in which a set occurrence selection clause may be expressed in the schema. We shall illustrate the general methodology of this selection process. When a set occurrence selection clause is included in a schema, a piece of code is generated which is called upon by the DBTGI if automatic selection of a set occurrence is required by the user program. Three possibilities are considered. In the following, a set occurrence will be identified by a triple consisting of the owner record type, L-number of owner record occurrence and its database key.

Case 1: Set Occurrence Selection through Current Record Occurrence of the Set

In this case, the set occurrence to be selected is the one that contains the current record of the set. For example, in Figure 5.2, the set type SETX has three occurrences X1, X2 and X3. The current record of the entire set is the record occurrence labelled r. Since r is contained in the set occurrence X2, the latter is therefore selected as the set occurrence for consideration.

The code to be generated for selecting a set occurrence through its current record is as follows:

(1) Let the set name be called set-type and the owner record type be called owner-type.

(2) From the currency indicator table extract the current set occurrence of set-type. Let it be identified by the triple (owner-type, owner-L-number, owner-db-key).

(3) The required set occurrence is (owner-type, owner-L-number, owner-db-key).

Case 2: Set Occurrence Selection through the Location Mode of the Owner

The selection strategy in this case is simple. Since it is necessary that the owner record type of the set be declared in the schema to have a location mode of either direct or calc, an occurrence of the owner record type can be identified by the run-unit with the database key or calc keys. Once the owner record is identified, the set occurrence of which the record is an owner becomes known. The code generated is:

(1) Let calc-keys represent the set of calc keys taken from UWA if location mode is calc; let db-key represent the database key, if location mode is direct.

(2) Determine owner-L-number from calc-keys or db-key, whichever is applicable.

(3) If location mode is direct, then the required set occurrence is (owner-type, owner-L-number, db-key).

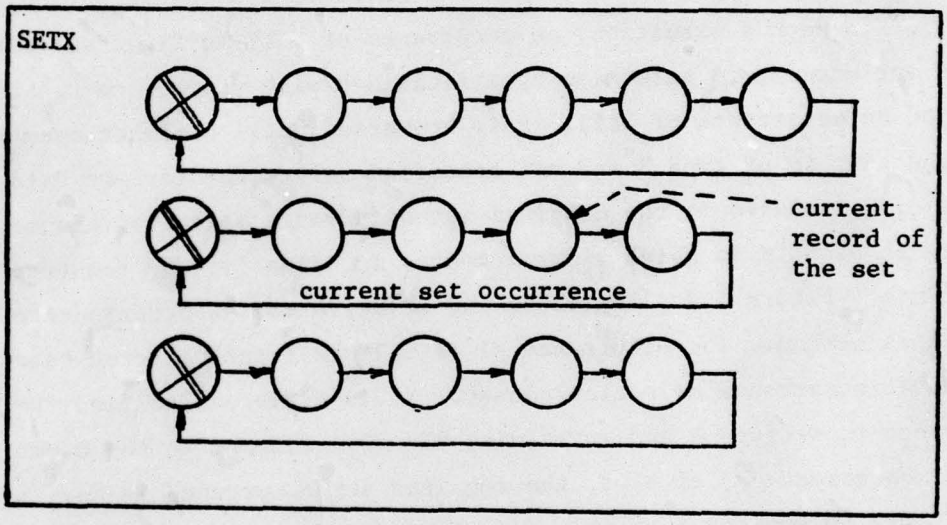


Figure 5.2. Set occurrence selection through the current record occurrence of the set

(4) If the location mode is calc, do the following:

i) issue command to DBC to "retrieve the record that satisfies <REC-TYPE=owner-type> ∧ <L-NUMBER=owner-L-number> ∧ calc keys";

ii) let db-key be the DB-key of the retrieved record. Then the required set occurrence is (owner-type, owner-L-number, db-key).

Case 3: Set Occurrence Selection through the Location Mode of the Owner using Database Identifiers

When the location mode of the owner record type R of a set SETX is neither direct nor calc but via another set SETY, it is not possible to directly select an occurrence of the owner record type. An occurrence of the owner record may then be selected on the basis of its participation as member in SETY. In such a situation, an occurrence of SETY is first selected using SETY's set occurrence selection specifications with respect to R. Having selected an occurrence of SETY, it is traversed until a member occurrence r is found that is of type R and has specified values for certain data items v1, v2,..., included in the original set occurrence selection clause. The occurrence of SETX in which r participates is, finally, the required set occurrence. Figure 5.3 illustrates the selection of an occurrence of SETX using this method. The occurrence Y1 of SETY is first selected based on SETY's set occurrence selection clause. Y1 is traversed to find the record occurrence r, of type R and satisfying v1, v2. Since r is the owner of the set occurrence X3 of SETX, the required set occurrence is X3.

It may be noticed that the selection of an occurrence of SETY may in turn, involve a reference to another set. Thus there may be a second level of indirection. The process can repeat to an arbitrary but finite number of levels. At the ultimate level, the set occurrence must be selected through the current record of the set or through the location mode of the owner, either direct or calc. Our code generation for set occurrence selection of the object set must follow the code generation for the sets that it depends on at any level of indirection. For example, the code to be generated for set occurrence selection of SETX through the location mode (via SETY) of owner (of type R), using data items (v1, v2,...,vn) is as follows:

(1) Copy the code for set occurrence selection of SETY with respect to its member record type R. This generates a set occurrence of SETY identified by, say, (owner-type-of-SETY, L-number-y, db-key-y).

(2) Retrieve the record satisfying (v1, v2,...,vn) by giving a

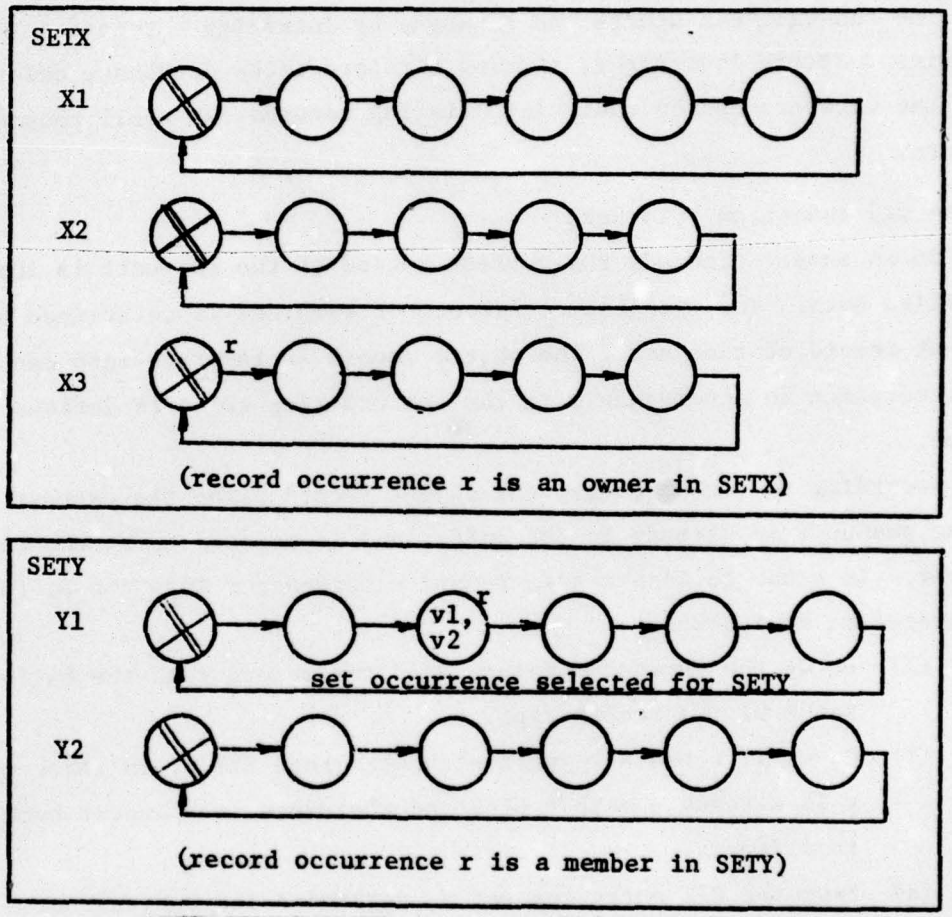


Figure 5.3. Set occurrence selection through the location mode of the owner using database identifiers

command to the DBC to "retrieve the record that satisfies <REC-TYPE=R> ∧ <L-NUMBER=L-number-y> ∧ <SET,SETY=db-key-y> ∧ <RECORD,v1=value-of-v1> ∧ <RECORD,v2=value-of-v2> ∧...∧ <RECORD,vn=value-of-vn>".

(3) Let db-key and LN be the database key and L-number, respectively, of the retrieved record. Then the required set occurrence is (R, LN, db-key).

5.2. Updating the Database

The run-unit can update the database by inserting a record into a set, removing a record from a set, storing a record inthe database, deleting a record from the database and by modifying existing records. We shall consider them in turn.

A. Record Insertion into Sets

On an insert command, the current record of the run-unit is inserted into specified sets. The specific occurrence of each set is determined by the current record of that set. The object record is inserted into each such set occurrence in accordance with the set ordering criteria defined in the schema.

According to our strategy, the object record being the current record of the run-unit is already in the buffer and is pointed to by the current-pointer. In order to insert this record occurrence r into set S, the following procedure is invoked.

(1) Using the current-pointer, find the record r in the buffer. Let R be its record type.

(2) Check if r has a keyword with attribute SET.S, in which case r is already a member of S and therefore, the insert command is illegal.

(3) From the CIT entry for set S, determine its current set occurrence. Let it be identified by the owner record occurrence (owner-type, owner-L-number, owner-db-key), which is referred as p.

(4) Retrieve the owner record occurrence p from the database by issuing a command to the DBC to "retrieve the record satisfying <REC-TYPE=owner-type> ∧ <L-NUMBER=owner-L-number> ∧ <DBKEY=owner-db-key>".

(5) Delete r and p from the database by issuing delete commands with their L-numbers, database keys and record types in the

query part.

(6) Modify r and p and store them back in the database. The necessary modifications are as follows. Adjust the keyword with attribute SET-MEMBER.S in p, so that it correctly indicates the current number of members. Adjust the keyword with attribute SET-SEQNUMBER.S in p if ordering is chronological. Include the keyword <SET.S, owner-db-key> in r, thus indicating its membership in set S. Include a keyword with attribute SETPOSITION in record r, if set ordering is first, last, next or prior. The value for SETPOSITION is determined as outlined in Section 4. In case re-numbering is necessary then the entire set occurrence is retrieved, deleted from the database, re-numbered in the buffer and inserted back in the database.

A general procedure to retrieve an entire set occurrence is given below. The identification of the set occurrence (owner-type, owner-db-key, owner-L-number) is provided to the routine. The given set type is S.

(1) From the schema entry for set S, determine the member record types. From the schema entry for each of these member record types, determine their location modes. Let $R_1, R_2, \ldots, R_n$ be the member record types whose location modes are via set S. Any occurrence of these record types will have an L-number which is the same as that of its owner. Let $R_{n+1}, \ldots, R_m$ be the other member record types of set S.

(2) The required set occurrence is now retrieved as follows.

(i) For each record type R_i, $1 \leq i \leq n$, issue a command to the DBC to "retrieve all records that satisfy <REC-TYPE=R_i> $\wedge$ <L-NUMBER=owner-L-number> $\wedge$ <SET.S=owner-db-key>".

(ii) For each record type R_i, $n+1 \leq i \leq m$, issue a command to the DBC to "retrieve all records that satisfy <REC-TYPE=R_i> $\wedge$ <SET.S=owner-db-key>".

In the above procedure, the entire set occurrence is retrieved, except the owner record occurrence. Normally, the owner record occurrence will already be present in the buffer. If it is necessary to retrieve the owner record occurrence as well, then issue an extra command to the DBC to "retrieve the record satisfying <REC-TYPE=owner-type> $\wedge$ <L-NUMBER=owner-L-number> $\wedge$ <DBKEY=owner-db-key>".

Storing a record back in the database involves the specification of its constituent keywords to the DBC. The DBC will then automatically update its directories and perform the appropriate clustering.

B. Record Removal from Sets

Removal of the current record r of the run-unit from a set S is straightforward. Its copy is already in the buffer and is located by using the current-pointer. Since r is a member of set S, it has two keywords <SET.S, owner-db-key> and <OWNER-L-NUMBER.S, owner-L-number>. These two keywords and the owner record type (got from the schema) may be used in a query to the DBC to retrieve the owner record of the set occurrence in which r belongs. The owner record is deleted from the database, its copy is modified in the buffer to correctly indicate its current number of members and it is stored back in the database. The object record r is also deleted from the database, its keywords related to set S (namely, the ones with attributes SET.S, OWNER-L-NUMBER.S and SETPOSITION.S) are removed, and the record is then stored back in the database. In the commands to delete the object record and its owner, the appropriate L-numbers and database keys are used in the query.

C. Deletion of Records from the Database

The current record of the run-unit may be completely deleted from the database by means of a delete statement. However, a deletion is usually associated with certain associated side-effects. Deletion of the current record of the run-unit may trigger a series of further deletions depending on the type of deletion requested by the run-unit. The delete statement can take one of three forms: delete-only, delete-selective and delete-all. A complete description of the significance of all these forms may be found in [2]. We, however, shall consider only the implementation of the delete-only operation which will serve to illustrate the other forms as well.

The delete-only statement deletes the object record r and all the mandatory members in the sets in which r participates as owner. The optional members are removed from these sets but are not deleted from the database. If any of the deleted mandatory members are themselves the owners of any set occurrences, then the delete statement is executed on such records as if they were the object records of delete-only statements. Thus all mandatory members of such records are also deleted, which in turn cause this process to continue down the hierarchy. Another effect of the statement is that the current record ofthe run-unit becomes null,

The execution of the delete-only statement requires a call to a recursive subroutine, delete-only, with the current record of the run-unit as argument. The subroutine calls itself whenever any further record is to be deleted. Prior to any call for delete-only with a record as argument, the record itself is first deleted from the database but its copy is retained in the buffer. A delete-only statement is, therefore,executed as follows:

(1) Using the current-pointer,find db-key, record-type and L-number of the current record of the run-unit.

(2) Issue a command to the DBC to "delete the record satisfying the query
<REC-TYPE=record-type> ∧ <L-NUMBER=L-number> ∧ <DBKEY=db-key>". Thus the current record is deleted but its copy is still in the buffer.

(3) Call delete-only with the current record of the run-unit as argument.

(4) Change the currency indicator of the run-unit in the currency indicator table (CIT) to null.

Procedure delete-only (r):

(1) Let db-key, LN and R be the database key, L-number and record type, respectively of the given record occurrence r.

(2) For each keyword <SET,set-name, OWNER> in r, do steps 3 through 6.

(3) Issue commands to the DBC to retrieve the set occurrence of set-name in which r is an owner. This is done as shown in part A of Section 5.2,

(4) Issue commands to the DBC to delete the set occurrence of set-name in which r is an owner, Thus the set occurrence is deleted but its copy is already in the buffer.

(5) For each of the records retrieved in step 3, if it is a mandatory member of set-name (as determined from the schema), then call delete-only with that record as argument.

(6) For each of the other records retrieved in step 3, remove the keywords related to the set set-name (in particular, those keywords with attributes SET.set-name, OWNER-L-NUMBER.set-name, SETPOSITION.set-name) and issue a command to store it back in the database.

D. Storing a Record in the Database

The user can store in the database a new record occurrence r of type R by first building up the data items in the user working area for R and then using the store statement. This statement also has the following effects. The new record occurrence r is made a member of appropriate set occurrences of those sets for which R has been declared to be an automatic member. Furthermore, r is made the current record of the run-unit, of the area in which it is stored, of the record type R and of all the sets in which it is an owner or an automatic member. Any of these currency updates may be suppressed by a qualified suppress command associated with the store statement. Finally, new set occurrences are established for each set for which R has been declared as owner.

The following steps are, therefore, necessary to store a record of type R in the database:

(1) For each data item name d with a value v as found in the UWA, create a keyword <RECORD.d, v>. Include the keyword in the record occurrence r.

(2) For each set S in which R is the owner, include the keywords <SET.S, OWNER> and <SET-MEMBER.S, 0>,which shows the count of the current number of members in the corresponding set occurrence owned by r. Unless the set is sorted, include also the keywords to indicate the position of r and the maximum (or minimum) position of any of its members in set S. The first of the two keywords is <SETPOSITION.S, 0>, if set ordering is first, last, next or prior. The other keyword is <SET-SEQNUMBER.S, m> if set ordering is first or last, where m is 0 if set ordering is last and m is a large number N if set ordering is first.

(3) Determine a database key and L-number for r based on the procedures outlined in Section 4. Include the corresponding keywords in r.

(4) For each set S in which R is declared an automatic member, determine an occurrence of set S by using the set occurrence selection clause (in schema) of S with respect to R. Insert r into this set occurrence as described in part A of Section 5.2.

(5) Update the currency indicator table (CIT).

E. Modification of a Record

Modification of a record can be done by first deleting it from the database and then storing it back in the database. Storing the record is done exactly as shown in part D of Section 5.2.

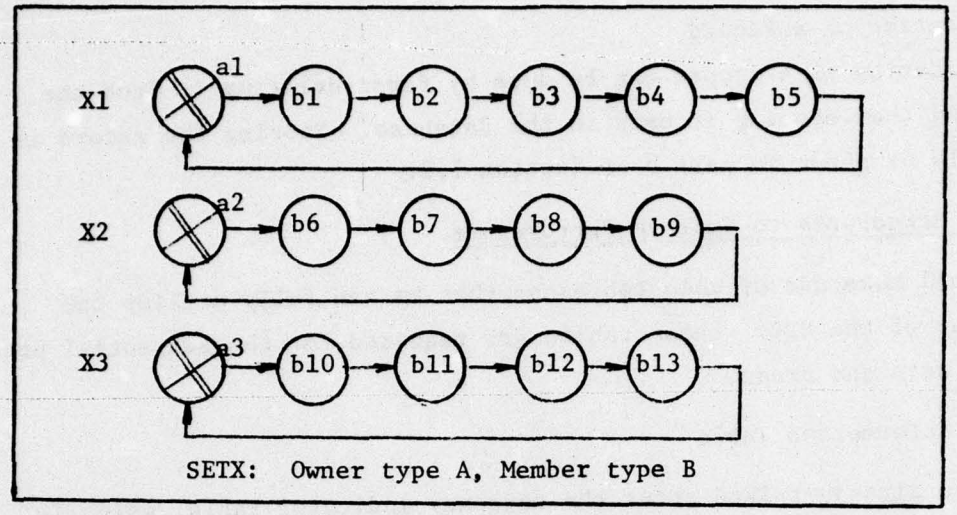
5.3. Data Structures to Improve Performance

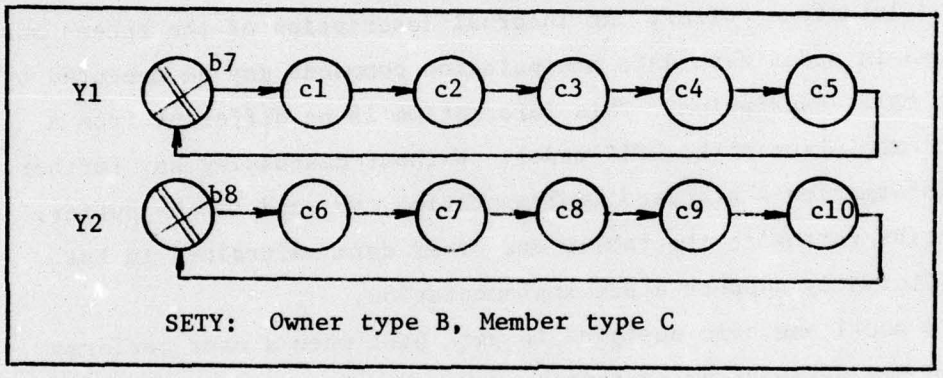
We shall make use of some tables so that we can fully utilize the capabilities of the DBC. These tables are required for the sequential processing of sets and areas.

5.3.1 Set Information Table

We have already talked about the currency indicator table, which is placed in the interface buffer. An internal description of the schema must also be stored in order that data manipulation commands may be executed by referring to this description. This information is no different from a standard implementation ofthe DBTG model. Without discussing any further what other information a standard implementation may need in the buffer, we will describe forthwith the tables and other data maintained in the buffer to efficiently support a DBC implementation.

The DBTG model has been designed to work best when a user performs operations on the database by navigating through sets. In fact, a great majority of the time, the user may be traversing one or more set occurrences in their entirety. A typical situation in which a user traverses through sets in a hierarchical order is illustrated in Figure 5.4 in which each of three different sets have only one member record type. SETX with owner type A and member type B has currently three set occurrences labelled X1, X2, and X3 in the database. SETY has only two occurrences and SETZ has three. The user may perform the following traversal routine. The set occurrence X2 is obtained directly by locating the owner record occurrence a2. X2 is now completely traversed by accessing every member bi in X2 and by traversing every occurrence Yj of SETY in which bi is an owner. While traversing an occurrence of SETY, in turn, every member ck in that occurrence is accessed and every occurrence Zm of SETZ in which ck is an owner is also traversed. Thus the traversal order is the following sequence of set occurrences: X2, Y1, Z1, Z2, Y2, Z3. In implementing this traversal sequence we will have in the buffer complete set occurrences, at most one of each set type. The buffer will have set occurrences as shown below at different stages of processing:





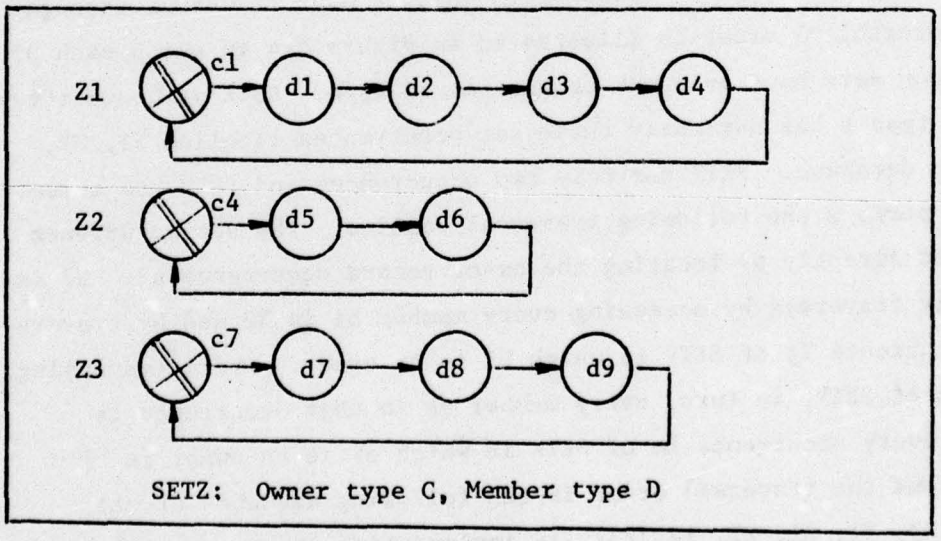


Figure 5.4. Example of set traversal

```
Stage 1.  X2          : process b6, b7
Stage 2.  X2, Y1      : process c1
Stage 3.  X2, Y1, Z1  : process d1, d2, d3, d4
Stage 4.  X2, Y1      : process c2, c3, c4
Stage 5.  X2, Y1, Z2  : process d5, d6
Stage 6.  X2, Y1      : process c5
Stage 7.  X2          : process b8
Stage 8.  X2, Y2      : process c6, c7
Stage 9.  X2, Y2, Z3  : process d7, d8, d9
Stage 10. X2, Y2      : process c8, c9, c10
Stage 11. X2          : process b9
```

The user traverses a set occurrence only after making it the current set occurrence. Thus at stage 5 of the above buffer configuration, X2, Y1 and Z2 are the current set occurrences of SETX, SETY and SETZ respectively.

In a general implementation, we may not have knowledge whether the current set occurrence of some set may be needed at a later time or not. It will be an unnecessary expense to keep a set occurrence of every set of the database in the interface buffer, if some of them are not to be referred again. Thus we shall undertake a policy of replacement (similar to the Least-Recently-Used policy of paging systems). We will detail this policy in Section 6 under the topic of buffer management. For the present, we will be interested in the tables necessary to keep track of the buffer information related to the set occurrences.

The set information table (SIT), will have an entry for every set type in the database, to indicate whether its current set occurrence resides in the buffer or not. The format of an entry of the SIT is shown in Figure 5.5.

The pointers are buffer addresses, where entire set occurrences are stored in consecutive locations. The member records are ordered as specified in the schema and the logical adjacency is reflected in physical adjacency. In case physical adjacency cannot be maintained, pointers (links) are employed. If the current occurrence of a set is in the buffer, then its owner record contains information on its database key and L-number. The fourth item of an entry in the SIT is useful when a user is interested in traversing the occurrences of only a particular member record type in a set occurrence, but is not concerned with the other member record types. In this case, the DBTG interface will retrieve only that part of the set occurrence which consists of member records of the specified type, thus preventing a wastage of buffer space and the time required to order unnecessary records.

Item	Description
1.	Set type
2.	Pointer to owner of current set occurrence -- null if current set occurrence is not in the buffer
3.	Pointer to current record of the set -- null if current set occurrence is not in the buffer
4.	Record type of member record -- valid only if all occurrences of a single member record type (and no other) of the current set occurrence is in the buffer
5.	Number of member records in the set occurrence as stored in the buffer

Figure 5.5. Format of an entry of the SIT

5.3.2 Set Traversal

A set is traversed by executing a sequence of find statements of the form

FIND NEXT (or PRIOR, or LAST, etc,) RECORD OF SET set-name

This statement retrieves the database key of the next or prior record with respect to the current record of the specified set, or it retrieves the n-th record (which may be first or last or any number) of the current set occurrence of the specified set. Another version of this statement also indicates a member record type (e.g., find the tenth record of type R in the current set occurrence of set S).

Because of the SIT and because of the fact that we retrieve entire set occurrences into the buffer, set traversal by using a succession of find statements of the above type will mostly involve sending a record from the buffer to the user. Only when the required set occurrence is not already in the buffer, must a command be sent to the DBC to retrieve that set occurrence.

To execute a find statement for traversal of set S, the following procedure may be invoked:

(1) From the currency indicator table (CIT) determine the current set occurrence of S. Let it be identified by (owner-type, owner-L-number, owner-db-key)

(2) Use the pointer of item 2 of the SIT (Figure 5.5) to determine if the current set occurrence is in the buffer.

(3) If the current set occurrence is in the buffer then go to step 5; otherwise, fetch the set occurrence by using the procedure described in part A of Section 5.2.

(4) Update the SIT to indicate the fact that the current set occurrence is now in the buffer.

(5) Find the new current record of the set. It is already in the buffer.

(6) Update the currency indicator table (CIT)

5.3.3 Sequential Processing of Records in an Area

Occasionally, a user may decide to process records sequentially in an area irrespective of the sets in which the records belong. This is an efficient way of processing a group of records if the order in which the records are handed out to the user is of no consequence. The DBTG system allows for this type of processing by letting the user request each record in an area one at a time. However, the records are actually given to the user in

an order determined by their database keys. This is the most natural way for a regular DBTG implementation because the records are actually stored in an order decided by database keys (even though database keys are defined to be logical identifiers and not physical addresses). In our implementation, records are clustered in MAUs by their L-numbers and not by area or database key. Since an area may be scattered over the entire database, sorting by database keys would imply storing an entire area in the interface buffer. A more logical method of sequential processing on the DBC is to access records sorted by L-numbers as the major key and the database keys as the minor key. Thus all records in an area are returned to the user in an order determined first by their L-numbers and secondarily by their database keys. This is a minor deviation from the DBTG definition but should really be of no concern to the user since he is primarily interested in sequential processing of records. However, if it is felt that the original definition (sorting by database keys alone) should be retained, then enough buffer space must be allowed to store all the records in an area. In our discussion, we shall assume that sequential processing in an area involves sorting, primarily, by L-numbers and, secondarily, by database keys. Consequently, we keep the following information as tables in the buffer.

AIT1 is a matrix that has an entry for each L-number and each area in the database. The value of AIT1 (A,L) is equal to the number of records in the database that belong to the area A and have an L-number L.

AIT2 is a one-dimensional array that has an entry for each area. An entry of AIT2 has the following information:

(1) Area name.
(2) Database-key of current record of the area.
(3) L-number of current record of the area.
(4) Pointer (in buffer) to the first record belonging to this area and having the above L-number ---- null if such records are not in the buffer.
(5) Pointer to current record of the area ---- null, if item 4 is null.
(6) Logical position of the current record of the area with respect to all the records of this area that have the L-number of item 3. (logical position is governed by L-number and database key).

The user may request the n-th record in an area. He may also request the next or prior record in an area with respect to the current record of that area. To answer such requests for an area A efficiently on the DBC, the following procedure is invoked:

(1) Determine the L-number LN of the requested record. If the request is for the n-th record, then the L-number is determined by using the area information table 1 (AIT1). If the request is for the next record or prior record with respect to the current record of the area, then the L-number is determined by using AIT2.

(2) If all records with L-number LN belonging to the area A are already in the buffer, then the requested record is also in the buffer.

(3) Otherwise, issue a command to the DBC to "retrieve all records that satisfy the query
<L-NUMBER=LN> ∧ <AREA=A>".
The DBC must be asked to sort these records by database key. The requested record is now in the buffer.

(4) Update table AIT2.

Suppose we hypothetically number all the records in an area A as determined by their L-numbers and database keys. To find the n-th record, then, we may use the table AIT1 to find the L-number L for this record and also its position among all the records of that area with the same L-number. Referring to table AIT2, we may check if that record is already in the buffer. If it is not in the buffer, then issue a command to the DBC to retrieve, in sorted order, all records that satisfy the query <L-NUMBER=L> ∧ <AREA=A>. Then update AIT2 by appropriately adjusting the pointers and record position in items 4, 5 and 6 of the entry for A.

AIT1 is updated easily whenever a new record is stored in the database. Since the L-number and area-name for a new record is known before it is stored, the entry in AIT1 corresponding to these subscripts is incremented by 1.

Notice that, for sequential processing of records in an area, at no moment do we need in the buffer more space than that required by all records with any given L-number. Since L-numbers are so chosen (or calculated) that all the records with any particular L-number usually occupy a memory size much less than an MAU size, our buffer storage requirement for sequential processing is not likely to exceed the MAU size.

5.4. Record Retrieval

There are many different ways in which the user may write a find statement in order to locate a record occurrence. Inspite of the variety, the

possible number of find statements may be classified into two categories. Either one finds a record occurrence based on its participation in a set (or area) or one finds a record occurrence based on knowledge of how it was initially stored in the database. We have already observed how a record occurrence is located based on its position in a set or area perhaps with respect to some other record occurrence that belongs to the same set or area.

The simplest type of find statement is the one in which the user specifies a record type R and a database key D. In this case the location mode of the record is direct. Retrieval of such a record can be done by using the query <REC-TYPE=R> ∧ <DBKEY=D>. However, records are clustered also by their L-numbers. Hence the above query may require a search of more MAUs then actually required although record type is an attribute of a type-D keyword. We, therefore, compute the L-number L of the record occurrence from its database key by hashing, as described in Section 4. The revised query, then, is <REC-TYPE=R> ∧ <DBKEY=D> ∧ <L-NUMBER=L>.

Another find statement requires locating the owner record occurrence of type R in set S, the occurrence of which is determined by the current record of a set X (or of a record type R_1 or of an area A or of the run-unit). The following steps may be performed to find the required owner record S.

(1) From the CIT entry for set X (or record type R_1, or area A or run-unit) extract the L-number L, the database key D and record type R_2 of the current record.

(2) Issue a command to the DBC to "retrieve the record satisfying the query
<REC-TYPE=R_2> ∧ <L-NUMBER=L> ∧ <DBKEY=D>".
This retrieves the current record occurrence r of set X.

(3) From record r, extract the keyword with attribute SET.S, whose value is, say, owner-db-key. Also extract the keyword with attribute OWNER-L-NUMBER.S, whose value is, say, owner-L-number.

(4) The required owner record occurrence of S is now retrieved by using the query
<REC-TYPE=R> ∧ <L-NUMBER=owner-L-number> ∧ <DBKEY=owner-db-key> .

Another type of find statement locates an occurrence of record type R, whose location mode is calc. The data items d1,...,dn specified in the location mode clause are initialized by the run-unit and stored in the UWA. The user may retrieve all occurrences of type R that have the same values for

the data items d1, d2,...,dn by executing a sequence of find statements of this type, but qualified by the next-duplicate-within clause; for example,

FIND NEXT DUPLICATE WITHIN RECORD TYPE R

Such a sequence of find statements can be very easily executed, and, in fact, in only a single access to the DBC. By hashing the data items d1,...,dn, determine an L-number L. Now issue a retrieval command to the DBC based on the query

<REC-TYPE=R> ∧ <L-NUMBER=L> ∧ <RECORD.d1=value-of-d1> ∧ . . . <RECORD.dn=value-of-dn>.

All the required records have the same values for L-number and the specified data items. Thus, they are all retrieved simultaneously. Only a single access will be required because of clustering based on record type and L-numbers.

Finally, we shall consider a find statement in which a set name S and the values of certain data items d1,...,dn are specified for a record type R. An occurrence of S is selected based on either the current record of the set or the set occurrence selection criterion for S. A member record occurrence of type R is now to be located that has given values for data items d1,...,dn. To execute such a statement, an occurrence s of set S is first selected. Let this occurrence be identified by the owner record with L-number L and database key D. The required member record occurrence in this set occurrence is found by using one of the two following queries depending on whether the location mode of R is via the set S:

(i) If the location mode of R is via S, then the L-number of the required member record occurrence is the same as that of its owner. Therefore, use the query

<REC-TYPE=R> ∧ < L-NUMBER=L> ∧ <SET.S=D>
∧ <RECORD.d1=value-of-d1> ∧ . . .
∧ <RECORD.dn=value-of-dn> .

(ii) Otherwise, the L-number of the member record occurrence is not known; so, use the query

<REC-TYPE=R> ∧ <SET.S=D>
∧ <RECORD.d1=value-of-d1> ∧ . . .
∧ <RECORD.dn=value-of-dn> .

This query will, in fact, retrieve all records of the set occurrence s that satisfy the specified data items. Thus any subsequent find statement requesting duplicates can be executed without further references to the database.

A user program may fetch the current record occurrence of the run-unit,

or parts of it, into the UWA by using a get statement. The DBTGI processes such a statement by simply transferring the record from the buffer into the UWA because the current record of the run-unit always resides in the buffer (possibly retrieved from the database in response to a find statement) and because the current-pointer points to this record.

To summarize our discussion in this section, we have tried to illustrate how DBTG commands are executed and what data structures are needed to support these commands in an efficient manner. We have shown how a member record can be retrieved from a knowledge of its owner and how an owner can be retrieved if the member record is given. We have also shown how entire set occurrences can be retrieved, how sequential processing can be done and how the database can be updated. Many of the procedures have been described only in outline in order not to burden the reader with excessive details. The level of abstraction has been chosen such that the basic design philosophy is expounded in a simple manner without sacrificing the consideration of important features of the DBTG model.

6. BUFFER MANAGEMENT

Like contemporary DBTG systems, which maintain system buffers for performance enhancement, the interface, i.e., the DBTGI also maintains a buffer space in order to enhance the performance of the DBC. Records from the database are retrieved by the DBTGI and stored in the buffer before any portion of this information is transmitted to the user working area (UWA). Although user commands may request records only one at a time, the DBTGI retrieves information in bulk. Thus, subsequent user commands need not necessarily cause the DBTGI to access the mass memory (MM). As long as the required record is in the buffer, it is directly sent to the user (or user's run-unit).

It is interesting to compare the buffer management policy of the DBTGI with one for a conventional DBTG system. The latter requires storage space both for buffering I/O blocks and for storing index tables. Index tables are partly system-determined and partly schema-determined. If a set is to be searched on specific data items for member records, then the conventional system must be induced to maintain index tables on these data items by including appropriate SEARCH clauses in the set entry of the schema. For singular sets (those sets of which the SYSTEM is the owner), the system normally maintains index tables to facilitate searching of the member records.

The buffer area in a conventional system is normally composed of two parts: the first part is a buffer work area for accommodating all the records that are absolutely necessary for carrying out a command such as delete, modify, insert, etc. Although the user may issue a command to process a single record, that command may trigger references to additional records in order to complete the processing (for example, the delete-only command described in Section 5). The other part of the buffer area is the I/O buffer. Because of the nature of secondary storage, such as disks, it is necessary for the conventional system to retrieve information in blocks; for example, a track. In response to a command, the address of a record is determined from the pointer field of another record in the buffer work area, or directly from the calc keys or from the database keys. If the block corresponding to that address is already in the buffer, then no I/O access is required; otherwise, that block must be retrieved into the I/O buffer and the contents of the addressed record must then be placed in the buffer work area.

In a conventional system, the records in a block may have diverse characteristics and need not always be logically related. When a block is

transferred from the database to the buffer, it is certain that one of the records in that block is of immediate use. However, the information content and characteristics of the other records in the block are not known. Thus, many of these records are likely to be irrelevant to the processing of the current or subsequent commands, thereby decreasing the effective buffer size and increasing the frequency of data transfer from the database to the buffer.

6.1 Buffer Organization of the DBTGI

The DBTGI does not require any space for index tables, but it does require buffer space to retain set occurrences and portions of an area. This buffer space, the interface system buffer (ISB), is designed to be implemented in a virtual memory environment as shown in Figure 6.1. Management of space in the ISB is done by the system buffer manager (SBM). When the DBTGI issues DBC commands for retrieving a group of records, the SBM requests the host operating system for virtual memory space for these records. Memory is assumed to be allocated in fixed size blocks, called pages.

For a given database, the DBTGI makes an estimate of the maximum buffer space requirement by referring to the average size of all set occurrences. Since the buffer primarily serves to enhance performance by storing a set occurrence of each of a few sets, the maximum buffer space estimate must be enough to accommodate four or five set occurrences. Whenever the DBTGI runs out of its virtual memory space, it throws out a set occurrence thus making place for another. During sequential processing, ISB also stores a portion of an area (a fraction of an MAU).

The SBM, when required to find virtual memory space, always requests enough space for a group of related records, such as records belonging to a given set occurrence or records with a given L-number and belonging to a given area. We shall refer to such a group of records as a logical block which may be stored in one or more pages.

Whenever the DML translator needs to refer to a record in a logical block, it first checks its own tables (e.g., SIT and AIT2 discussed in Section 5) to determine if the record is already in the buffer or not. If the record is already in the buffer then no further DBC access is necessary.

There are basically two types of DML statements affecting the DBTGI buffer requirements. First, the find statements that are used by a run-unit in order to traverse an area or a set; for example, find next, find prior, etc. The other, non-find, statements include modify, insert, delete, get, etc. The DBTGI normally expects the object record of a find statement to be present in the ISB. Only if it is not there is a reference made to the DBC

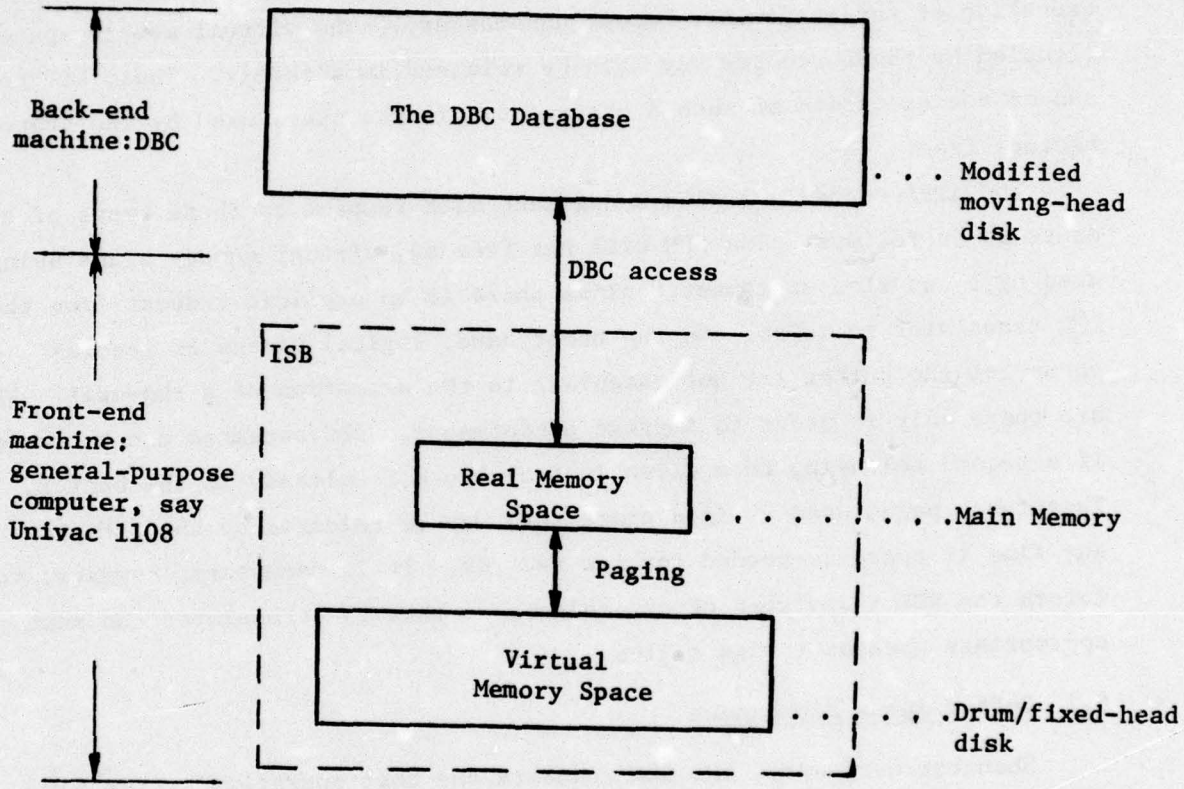


Figure 6.1. The virtual memory environment of the ISB

to retrieve a logical block of records. This block may occupy a number of fixed-size pages in the virtual memory, and its information content is accounted for by making an entry in the set information table SIT or in the area information table AIT2. Non-find statements may trigger accesses to a sequence of logical blocks. In this case, no information is assumed to be present in the ISB. Groups of records, as they are required, are fetched from the database and stored in the ISB. A logical block of records fetched into the ISB in response to such a statement will, at some stage during the execution of the statement, become unnecessary. The virtual memory space occupied by these records may then be released immediately. Thus, at the end of the execution of such a statement, all the space used by the statement becomes free.

The difference in buffer management with respect to these types of statements is as follows. The SBM will not free any virtual memory space being used by a non-find statement, unless there is an explicit request from the DML translator to do so. On the other hand, logical blocks of records occupying the buffer are not essential to the execution of a run-unit. They are there only in order to improve performance. DBC accesses can be saved if a record belonging to a given logical block is already in the buffer. Therefore, pages used by find statements may be released by the SBM at any time if space is needed for new records. It is necessary, however, to inform the DML translator of any release so that the translator can make appropriate updates to its tables.

6.2 Buffer Space Management

Whenever necessary, the SBM requests the host operating system for additional pages of virtual memory space. These pages are kept by the SBM in a doubly-linked list. If the operating system is unable to provide more space, then the SBM releases some space to the operating system. The SBM releases only those pages that are occupied by a logical block of records used in the processing of a find statement. In addition, in choosing such a block of records, a least-recently-used policy is employed. The idea is to throw out a block of records that has been referenced least recently.

Information on the logical blocks occupying the buffer is kept in a table called the logical block control table (LBCT). There is an entry in this table for every logical block of records currently occupying the buffer. Each entry of LBCT has the following fields corresponding to a logical block:

(1) FPADDR : address of the first page used by the logical block; used for forward traversal of the block.

(2) LPADDR : address of the last page used by the logical block; used for backward traversal of the block.

(3) TYPE : the type of statement that requested this block; either 'find' or 'non-find'.

(4) TIME : the time at which the block was last referenced; generated from the system clock.

(5) NAME : name of the area or set type to which this logical block belongs; used by the DML translator in making updates to its tables when the logical block is thrown out of the buffer.

(6) USE : a control field of one bit indicating whether this LBCT entry is currently in use or not.

LBCT is a sequential table of a very few entries, say at most 100, because it is expected that no more than 100 logical blocks will ever reside in the buffer simultaneously. Whenever a new block is to be entered in the buffer, a roving pointer, called ROVER, is incremented until a free entry is found in LBCT. The increment is modulo the maximum number of entries in LBCT.

Each page in the buffer has three pieces of control information in addition to a part of the contents of the logical block using that page. These three pieces of information are

(1) FPTR : forward pointer to the next page used by the logical block.

(2) BPTR : backward pointer to the previous page used by the block.

(3) NREC : number of records stored in this page; used, for example, in finding the n-th record in the block.

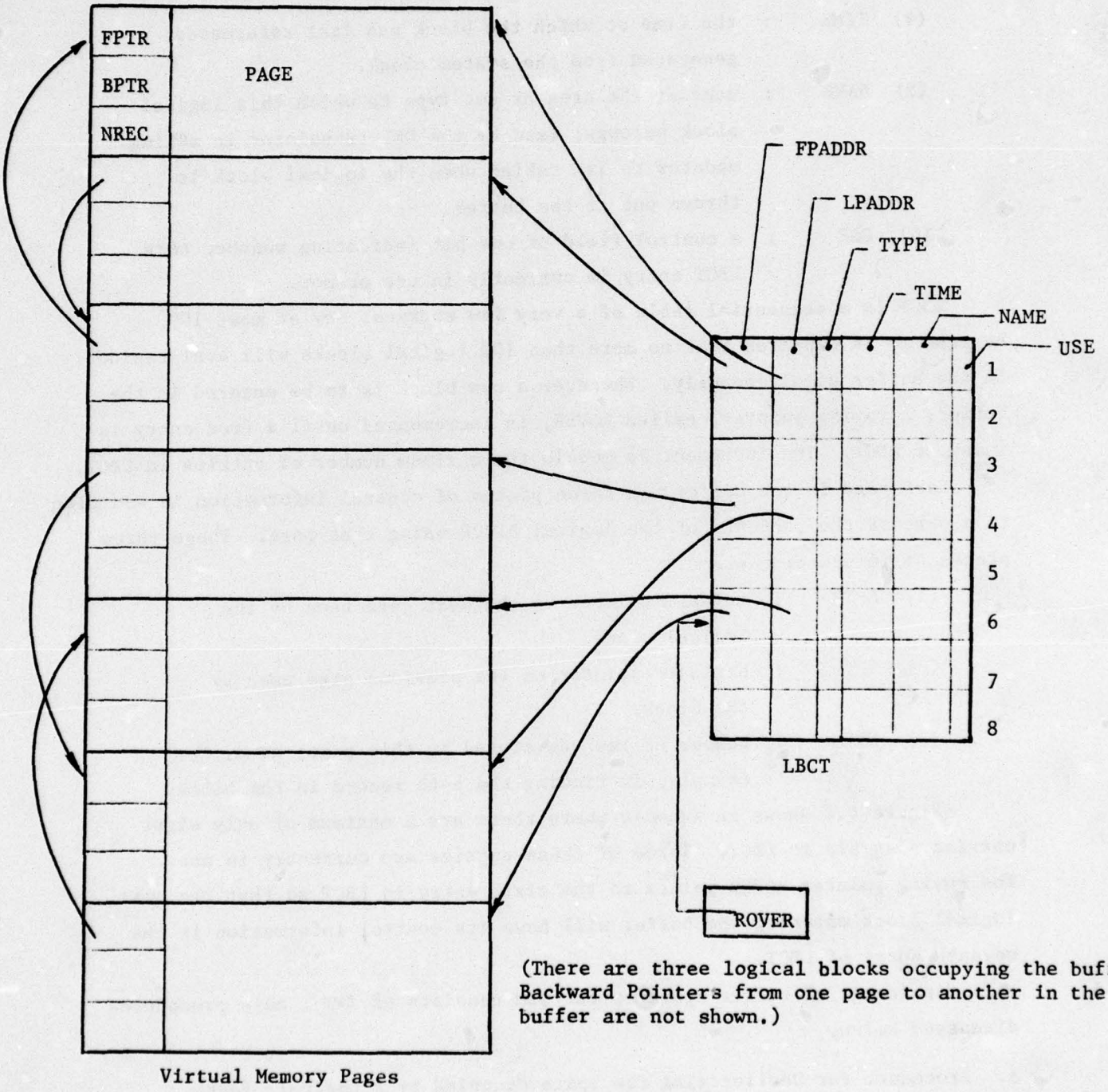
Figure 6.2 shows an example where there are a maximum of only eight entries possible in LBCT. Three of these entries are currently in use. The roving pointer ROVER points to the sixth entry in LBCT so that the next logical block entering the buffer will have its control information in the seventh entry of LBCT.

Management of buffer space by the SBM consists of two main procedures discussed below.

A. Procedure for Deallocating the Space Occupied by a Logical Block

Given a number n, this procedure releases the space occupied by the logical block corresponding to the n-th LBCT entry.

(1) Traverse the chain of pages starting at the page indicated by the FPADDR field of LBCT[n] (where LBCT[n] refers to the



(There are three logical blocks occupying the buffer. Backward Pointers from one page to another in the buffer are not shown.)

Figure 6.2. Relationship among ROVER, LBCT and virtual memory pages

n-th entry of LBCT). Return each of these pages to the operating system.

(2) Turn off the USE bit of LBCT[n].

B. Procedure for Allocating Space to a New Logical Block

This procedure allocates m pages to a new logical block used by a statement of type T. The name of the area or set type to which the block belongs is X.

(1) Request m pages from the operating system.

(2) If the request is successful then do the following:

(i) Increment ROVER (modulo the table size of LBCT).

(ii) Turn on the USE bit of LBCT[n]. Enter the current time in the TIME field, X into the name field, and T into the TYPE field of LBCT[n]. Enter the addresses of the first page and the last page in the FPADDR and LPADDR fields of LBCT[n].

(iii) Chain together the m pages and enter the records in those pages.

(iv) Return the value of n to the DML translator and stop.

(3) If the request is unsuccessful, then scan LBCT for an entry with the USE bit on, whose TYPE field has a value of 'find' and whose TIME field is the minimum (least recently used). Using the previous deallocation procedure, return the pages corresponding to this entry to the operating system. Go back to step 1.

Notice that instead of providing the address of the first page of a block to the DML translator, the SBM provides it with the entry number in LBCT corresponding to that block. The DML translator can now refer to the records in a logical block via the LBCT entry. And whenever it does so, it adjusts the TIME field in that entry so that the TIME field always reflects the most recently referenced time.

7. EVALUATION OF THE INTERFACE MODULE (DBTGI)

Our discussion of the interface module will not be complete unless we can demonstrate its low complexity and high performance. In this section, we shall make an attempt to analyze the interface module and evaluate its memory and access time requirements against a conventional implementation of the DBTG model. While there is some agreement as to the relevance of these two criteria, there has not been any outstanding work in the performance study of database management systems. The difficulty arises due to the tremendous variability of the size, content and organization of the databases. To complicate matters, the variety of user requirements makes the evaluation of data models and their associated systems difficult. Since the users may be more concerned with quick installation, they may be apathetic to change. Once accustomed to using a particular data model, they may resist the installation of a new model. So the criteria for evaluation are not merely the memory requirement and response time, but also the simplicity of the system and general user acceptance.

Simulation is perhaps the most common method of evaluating a system. Analytical results on an entire system at different levels of detail are impossible especially for a large database management system. But simulation also is very tedious and difficult. A simulation model must unify all the diversities of a system. In addition, it must be based on specific knowledge of user behaviour in terms of the types and order of queries, characteristics of the database, etc.; and such statistics may be difficult to obtain.

In the light of the above discussion, we shall ask ourselves to conduct a comparative study by making an operation-by-operation analysis of the two systems involved. The idea is that if each part of one system works as well or better than the comparable part of the other system, then it is plausible that the former will perform better than the other as a total system. This approach can provide an insight into the type of differences that arise in using the DBC v.s. a conventional computer for implementing the DBTG model.

7.1 Memory Requirement

The on-line storage cost of the DBC is comparable to that of a conventional system, such as DMS 1100 [8,9]. We shall point out that the DBC requires slightly more secondary storage but DMS 1100 requires more primary memory for storing and manipulating indexes. The directory memory requirement of the DBC is extremely small. Besides, DBC directories are stored in the structure memory (SM) and no extra space is required in the front-end computer to

manipulate these directories.

A record stored in DMS 1100 consists of a database key, the values of data items and some pointers. One or two pointers are required (depending on the type of linking) for every set in which the record participates as a member or owner. On the other hand, every record stored in the DBC consists of a combination of attribute-value pairs. There is such a pair for every data item, one for the database key, one for the record type, one for area name, three for every set in which the record is a member and three for every set in which the record is an owner. Now, some of the attributes consist only of one part, namely, the special word REC-TYPE, AREA, DBKEY or L-NUMBER. The attribute related to a data item consists of two parts: the special word RECORD and the name of the data item. An attribute related to a set also consists of two parts: one of the special words SET, OWNER-L-NUMBER, SETPOSITION, SET-MEMBER or SET-SEQNUMBER and the name of the set. Since there are only ten special words in all, they can be coded by 4 bits of information. If there are R record types, A areas, S set types and I data items for each record type, then they can be coded by $\log_2 R$, $\log_2 A$, $\log_2 S$ and $\log_2 I$ bits respectively. In a practical database, we do not expect any of these four numbers to exceed 4,096. Thus, they can be coded by 12 bits, and the attribute part of any keyword will never exceed 4+12=16 bits.

The database storage requirements for a single record in the DBC and in DMS1100 are given in Table 7.1. The unit of storage is assumed to be a byte (8 bits). We assume a size of 4 bytes for database keys, data item values, sequence numbers, pointers, etc.

Hence, if a record has r data items and participates in s sets (as either an owner or a member), then, the storage requirements are:

for DBC, $14 + 6r + 18s$ bytes

for DMS 1100, $4 + 4r + 4s$ bytes(if sets are singly-linked)

or $4 + 4r + 8s$ bytes(if sets are doubly-linked).

As an example, if a record participates in two sets and has 5 data items, then it is represented by 56 bytes on the DBC and either 32 or 50 bytes in DMS 1100. In general, the storage ratio S_m is

$$S_m = \frac{7+3r+9s}{2(1+r+s)} \quad \text{if sets of DMS 1100 use single links}$$

$$\frac{7+3r+9s}{2(1+r+2s)} \quad \text{if sets of DMS 1100 use double links.}$$

For these extreme cases,

(i) if $r = s = 0$, then $S_m = \frac{7}{2}$

Record	DBC		DMS 1100
Information to be stored	Keyword	Storage Requirement (in bytes)	Storage Requirement (in bytes)
Record type	<REC-TYPE, record-type>	2	0
Area	<AREA, area-name>	2	0
Database key	<DBKEY, db-key>	1 + 4 = 5	4
L-number	<L-NUMBER, L-number>	1 + 4 = 5	0
For each data item	<RECORD.data-item, value>	2 + 4 = 6	4
For each set membership	<SET.set-type, owner-db-key>	2 + 4 = 6	4 or 8
	<OWNER-L-NUMBER.set-type, owner-L-number>	2 + 4 = 6	0
	<SETPOSITION.set-type, sequence-number>	2 + 4 = 6	0
For each set ownership	<SET.set-type, OWNER>	2 + 4 = 6	4 or 8
	<SET-MEMBER.set-type, n>	2 + 4 = 6	0
	<SET-SEQNUMBER.set-type, m>	2 + 4 = 6	0

Table 7.1. Storage requirement for a single record

(ii) if r>>s, then $S_m \simeq \frac{3}{2}$

(iii) if s>>r then $S_m \simeq \frac{9}{2}$ or $S_m \simeq \frac{9}{4}$.

In general, it may be concluded that the DBC mass memory requirement is approximately one and a half times that of DMS 1100, because in most practical cases r > s.

However, the excess cost in mass memory requirement is amply compensated by the avoidance of major tables in the main memory. In DMS 1100, there is an index table on every data item which is specified as a search key in the schema. A set may be searched for member records with specified search keys, and, in the absence of indexes, these searches will be expensive in terms of access time. Therefore, since there is a wide variety of ways in which a set may be searched, indexes may have to be maintained on many data items. The DBC has directories on record types, area names and L-numbers. These directories are stored in the structure memory and are maintained automatically by the DBC. Thus no storage space is required in the front-end computer for software maintenance of these directories. In addition, since the number of record types, areas and L-numbers is very small compared to the number of values for various data items, the DBC directories are generally much smaller than the DMS 1100 indexes.

In a somewhat simplified analysis in Section 5, we have seen that a large database of 10,000 MAUs, approximately 10^{10} bytes, requires a directory size of less than 10^6 bytes. Even if we assume a case in which the directory is ten times larger than this estimate, it still does not exceed 0.1% of the database size.

In contrast, consider the index memory requirement in DMS 1100. In keeping with practical network databases, assume that the bulk of the records belong to at least one set. Assume, further, that every set is indexed, on the average, on one data item belonging to its member records. Thus, for every record, there is at least one index entry. If an index entry is 8 bytes (4 bytes for a data item value and 4 bytes for a database key or pointer) and if the average record is 100 bytes long, then the size of the index is about 12% of the size of the database. Thus the index memory requirement in DMS 1100 is likely to be at least 100 times more than in the DBC.

The buffer size of DMS 1100 need not exceed the size of two or three fixed size blocks (or pages), since records are needed only one at a time but are retrieved in pages. On the DBC, records are retrieved by content and

the number of such records retrieved together will be the largest when an entire set occurrence is brought into the buffer. Sets are usually not traversed more than a few at a time. Thus, the buffer need not store more than a few set occurrences. We conclude that if the size of an average set occurrence is five times the page size of DMS 1100, then the DBC buffer storage requirement will be about five times that of DMS 1100. However, every record in the DBC buffer is a "valid" record in the sense that part of its content is known to the DBTGI and can therefore be used in processing subsequent DML statements. The contents of a DMS 1100 page, on the other hand, are unknown to the DMS 1100 system.

7.2. Analysis of Access Time Requirement

An important measure of performance is the efficiency with which DML programs are executed. In this section, we shall compare execution rates by estimating the number of mass memory accesses required to perform various operations. With an approximate knowledge of the relative frequency of these operations, the overall performance ratio can then be estimated.

A. Finding a Record Based on Database Key

If the location mode of a record type is direct, then it may often be necessary to locate one of its instances through its database key. In DMS 1100, the database key consists of an area name, a logical page number and a record number. An <area, page> index is used to locate the physical page (or block) corresponding to the area and logical page number. The required record must reside in this page; thus enough space must necessarily be available in the page while storing the record. Thus, a single access will suffice.

In the DBC, the database key is used to derive an L-number; and a record is found that satisfies this L-number, the record type and the given database key. Since clustering is done on record type and L-number, it is extremely unlikely that more than one database access will be necessary to locate the record. However, while direct record placement in the DBC is data independent, that is not the case with DMS 1100. To determine a database key correctly, the DMS 1100 user must have exact knowledge of the records that have already been placed in the same page.

B. Finding a Record Based on Calc Keys

In DMS 1100, given the calc keys, a procedure is invoked to produce a page number and a calc-chain number. The procedure determines these numbers by means of a table look-up, or by key transformation, or by simply incrementing a special counter. The calc record placement logic will place the object

record on the data page indicated by the page number, as long as sufficient space exists in that page. In case of insufficient space, an overflow page may be used and chained to the original data page. All records that produce the same page and calc-chain numbers will be in the same calc-chain but may spread over an initial data page and one or more overflow pages. Thus, to find a record from the calc keys, usually one access, and sometimes more, will be necessary. In addition, a table is required on every page for each calc-chain beginning in that page.

In the DBC, an L-number is determined from the calc keys. The L-number, together with the record type, automatically identifies an MAU number (since a directory is maintained on record types and L-numbers). An associative search using the given calc keys will now find the required record. Thus, a single database access is sufficient.

To find duplicates or all records having identical calc keys requires the same single access on the DBC since they all have identical L-numbers and are of the same record type. In DMS 1100, however, the entire chain corresponding to the given calc keys must be traversed, requiring one or more page accesses.

C. Record Retrieval Via Sets

The single most important group of operations performed on a network database are the operations involving the manipulation of records based on their participation in sets. In such an operation, a record must first be located by an appropriate find statement. We shall soon discover that when sets are being manipulated, the DBC attains the maximum gain in access time as compared to DMS 1100. Since set manipulation constitutes a major portion of a run-unit, the use of the DBC in implementing a network data model will be rewarding.

In DMS 1100, a page is the unit of retrieval from the secondary storage and usually corresponds to a track in a disk unit. In the DBC, an entire MAU can be searched in one access, and the size of an MAU corresponds to the size of a cylinder of a disk unit. A realistic assumption is that there are 40 tracks per cylinder. Therefore, the DBC can access as much as 40 times more information than can DMS 1100 in any given period of time. Assume, furthermore, that all records are equal in size, 100 bytes in the case of DMS 1100 and between 150 and 200 bytes in the case of the DBC (since DBC records are about fifty percent larger, as we have seen in Section 7.1). We shall take the worst case estimate for the DBC and assume a record size of 200 bytes. Let a page size in DMS 1100 be 10,000 bytes, thus accommodating as many as

100 records in a page. The MAU size in the DBC is 40 times the page size of DMS 1100; thus, an MAU can accommodate 2,000 records. We assume one member record type per set type since, in practice, this is most common. Let

M = total number of occurrences of a member record type; we will consider M = 10, 100, 1000, 10000, and 100000.

n = number of member record occurrences in a set occurrence; we will consider n = 1, 10, 100, 1000 and 10000.

e = clustering efficiency (to be defined shortly); possible values considered are 0.1, 0.2, 0.3, 0.4 and 0.5.

DMS 1100 tries to cluster all member record occurrences of a set occurrence into as few pages as possible if the member records have been declared in the schema to have a location mode via that set. In the DBC, all occurrences of any record type are clustered into as few MAUs as possible. In addition, all occurrences of a given record type that have the same L-number are further clustered. However, both in the case of DMS 1100 and the DBC, clustering may not always be perfect because of insertions and deletions from the database. Therefore, we introduce an efficiency measure e, which is the ratio of the number of physical blocks (pages in DMS 1100, MAUs in the DBC) absolutely required to store a cluster, to the number of physical blocks actually used to store the cluster.

Let us consider, now, the problem of finding a member record in a set occurrence. Given the identification of the set occurrence, it is required to find a member that satisfies given values for certain arbitrary data items. If a directory is available on these data items, then the number of accesses required will be small, at the cost of main memory space. However, if the diversity of values of the data items is large, then a directory is almost infeasible. Besides, if the data items required to be compared are totally arbitrary, then a directory cannot be expected. We, therefore, analyze the problem with an assumption that directories are not available on the data items.

There are two cases:

(1) the member record type is declared to have a location mode via the set under consideration,

(2) the member record type is declared to have a location mode via some other set or direct or calc.

Case 1. Location Mode is via the given set.

In this case, the set occurrence is clustered into as few blocks as

possible. In DMS 1100, it may be assumed that it will be necessary to travel half-way through the set occurrence before the required record is found. In the DBC, all the MAUs corresponding to the set occurrence are associatively searched by hardware. All the members of the set occurrence have the same record type and L-number and are therefore clustered.

In DMS 1100, since there are 100 records per page and the set occurrence has n members, the entire set occurrence will be accommodated in $\left\lceil \frac{n}{100} \cdot \frac{1}{e} \right\rceil$ pages, where e is the clustering efficiency. Assuming that the member records are stored in such a way that it is not necessary to retrieve a single page twice while traversing a set occurrence, we need $\left\lceil \frac{n}{200e} \right\rceil$ accesses to find a record half-way through the set occurrence.

In the DBC, since there are 2,000 records per MAU, $\left\lceil \frac{n}{2000e} \right\rceil$ MAUs will be accessed. Notice that when n is large or e is small there are about 10 times as many DMS 1100 block accesses as DBC MAU accesses. In addition, the DBC reduces the CPU processing time as well since a hardware associative search is performed rather than a software search. In Table 7.2, we have tabulated the number of accesses corresponding to various values of n and e.

Case 2. Location Mode is Not Via the Given Set

When the location mode of the member record is not via the given set, then it cannot be assumed that member records of a set occurrence will be clustered. In fact, the member records are likely to spread over the entire database. However, in the case of the DBC, since clustering is primarily by record type, the member record occurrences, being all of the same type, will be clustered with all other occurrences of the same record type.

In DMS 1100, since a set occurrence consists of n member records, an average of $\frac{n}{2}$ records will have to be searched before the appropriate record is found. Since the records are not clustered, every record access will usually require a page access so that $\frac{n}{2}$ pages will have to be accessed.

In the DBC, it will be sufficient to search all occurrences of the given record type in order to find the right record. Because all these records are clustered and there are M occurrences per record type, the records will be clustered in $\left\lceil \frac{M}{2000e} \right\rceil$ MAUs, and only these MAUs need be accessed. In Table 7.3 we have tabulated the number of accesses required by DMS 1100 and by the DBC. The ranges of values chosen for n and M are realistic. Thus, in general, we can conclude from this table that the DBC requires at least an order of magnitude fewer accesses than does DMS 1100. On the average, perhaps, DMS 1100 will require about 50 times as many accesses as the DBC, since the clustering efficiency of 0.5 should be easily attainable. As an additional advantage,

n = number of member record occurrences in a set occurrence
e = clustering efficiency

e \ n	1	10	100	1000	10000
.1	1	1	5	50	500
.2	1	1	3	25	250
.3	1	1	2	17	167
.4	1	1	2	13	125
.5	1	1	1	10	100

(a) number of page accesses in DMS 1100

e \ n	1	10	100	1000	10000
.1	1	1	1	5	50
.2	1	1	1	3	25
.3	1	1	1	2	17
.4	1	1	1	2	13
.5	1	1	1	1	10

(b) number of MAU accesses in the DBC

Table 7.2. Number of block accesses (for various values n and e) to find an arbitrary member record in a set occurrence when location mode is via that set

e = clustering efficiency
n = number of member record occurrences in a set occurrence
M = number of occurrences of a member record type

n	1	10	100	1000	10000
accesses	1	5	50	500	5000

(a) number of page accesses in DMS 1100

e \ M	10	100	1000	10000	100000
.1	1	1	5	50	500
.2	1	1	3	25	250
.3	1	1	2	17	167
.4	1	1	2	13	125
.5	1	1	1	10	100

(b) number of MAU accesses in the DBC

Table 7.3. Number of block accesses to find an arbitrary member record in a set occurrence when location mode is not via that set

once again, the software search of DMS 1100 is replaced by an associative hardware search in the DBC.

Extending the study a little further, suppose that we are to find duplicates on certain data items. In other words, it may be necessary to find all member records in a set occurrence that have the same values for specified data items. In such a situation, the number of accesses required on the DBC is unchanged from our previous example. If the location mode of the records is via the given set then the number of DBC accesses are as shown in Table 7.2(b). If the location mode is not via the given set then the number of DBC accesses are as shown in Table 7.3(b). But in the case of DMS 1100, it may now be required to traverse the entire set occurrence instead of only half of it, doubling the page accesses and thus doubling the entries in Tables 7.2(a) and 7.3(a). Therefore, if the location mode of the member record type is not via the given set, then the DBC performance is likely to be 100 times as good as that of DMS 1100.

D. Set Traversal

A very frequent operation on sets is the traversal of a set. Since sets are the only link between two different types of records, all inter-record type operations will require the traversal of sets. The system also will often have to navigate through sets without any direct user command. For example, while storing an automatic member record occurrence in the database or during the modification of certain data items in a record, it may be necessary to insert the record in a new set occurrence. The selection of the appropriate set occurrence has to be done automatically by the system, using the set occurrence selection criterion. As we have discussed in Section 4, there are many ways of selecting a set occurrence, some of which involve (recursively) traversing one or more sets. The remarks we have just made are only to illustrate the fact that set traversal is indeed a very frequent phenomenon.

It is easy to observe that analysis of access times for set traversal is identical to what we have discussed for record retrieval via sets. The number of DBC accesses are as shown in Tables 7.2(b) and 7.3(b). DMS 1100 requires double the number of accesses shown in Tables 7.2(a)and 7.3(a). Thus the DBC performance in set traversal will be usually between 10 and 100 times better than DMS 1100.

E. Other Operations

The other operations of importance are insertion of a record into a

set, removal of a record from a set, storing a record in the database and deleting a record from the database. Most of these operations are usually preceded by initializations of the currency indicators of sets, record types and the run-unit. Thus, once again, set traversals may be necessary, but in this section we only consider the specific operations of insertion, removal, deletion and record storage.

In DMS 1100, the insertion of a record into a set involves access to the record to be inserted as well as the prior record in the set, in order to adjust its link field. If the sets are doubly-linked then another record, the one next to the inserted record, must also be accessed and its link field modified. Thus, either two or three records must be accessed to conduct an insertion operation.

In the DBC, if the set ordering is first, last or sorted then only two records need be accessed (the owner record and the inserted record). In case the set ordering is next or prior, then the entire set occurrence can be retrieved in 1 to 10 accesses, if clustering efficiency is 0.5 (refer to Table 7.2(b)).

The removal of a record from a set requires access to only a single record in the case of the DBC since there are no pointers to adjust. In DMS 1100, however, up to three records will have to be accessed if sets are doubly-linked.

The deletion of a record from the database, as we have observed in Section 5, may involve accesses to a series of set occurrences when it is an owner record. We have already noted that set occurrences are accessed or traversed at least an order of magnitude faster in the DBC than in DMS 1100. In particular, we have also seen that the speed in traversing a set occurrence can be improved by a factor of 100. Besides, all records retrieved during this traversal process are either deleted or modified. Finally, the deletion of an entire set occurrence from the DBC is done by a single command to delete all records that satisfy the query consisting of the identification of the owner of the set occurrence and the set name. In DMS 1100, on the other hand, deletion of each record of the database is done only one at a time.

The operation of storing a record in the database requires a single access in the case of the DBC. In DMS 1100, it will require only one page access if location mode is direct and one or more page accesses if location mode is calc. While storing a record it may be required to insert the record into one or more sets. Once again, if these insertions involve set traversals then the DBC turns out to be an order of magnitude faster than DMS 1100.

7.3. Summary of the Evaluation

In the absence of a group of representative network programs, it is not possible to make an analysis or a simulation experiment that will reflect the performance of the DBC or DMS 1100 with absolute precision. Our approach in this section was to make an operation-by-operation analysis to quantitatively and qualitatively demonstrate the superiority of the DBC. The greatest gain in performance has been achieved in set manipulation, the basic operation in the network data model. We therefore may expect between 10 to 100 times performance improvement if the DBC is used to replace a conventional computer system and software.

The main reasons for improved performance are that, first, an MAU is about 40 times as big as a page and, second, the clustering policy allows all occurrences of a record type to be placed together. Since the DBC can retrieve as much as one MAU rather than a page of valid information in one access, it is a much faster machine. In a regular DBTG implementation, the placement of records is guided by their database keys. Thus set occurrences tend to scatter over many pages. Our policy of clustering first by record type and then by L-number ensures that all members of a set occurrence will always lie close to one another.

The reason why it is possible to access an entire MAU is the hardware associative search capability. Without resorting to having a very large I/O buffer, it is possible to access information in bulk because records are retrieved by content and therefore only those records that satisfy a given query will be retrieved. No records which are unknown or unaccounted for will spill into the buffer, as would happen if the memory were not content-addressable. Furthermore, no "pointer-pushing" is necessary in the main memory since we have not allowed pointers in the DBC records. Consider, for example, the problem of retrieving a record from a set, with specified data item values. In the DBC, the record (and duplicates, if any) will be directly retrieved by content. No storage space will be required in the front-end computer for conducting any software search. CPU cycles will also be freed up since software search is not necessary. In the case of DMS 1100, it may be necessary to travel halfway through the set occurrence, and this, as we have seen in Table 7.2(a), may require 10 page accesses, if the clustering efficiency is 0.5 and if there are 1000 records in the set occurrence. But the approximately 100 member records within a page must be searched in the main memory. Thus, for each page accessed, 100 pointer-pushing operations are required in DMS 1100.

Furthermore, storage space will be required to conduct a software search for the given data item values.

In traversing a set occurrence, the DBC requires retrieval of the set occurrence in a sorted order. But then, this sorting is done by hardware and should not be very time-consuming. DMS 1100 will require sorting during the time of creating the database and inserting a member record occurrence into a set. It will, further, require many page accesses to retrieve all the member records of a set occurrence.

8. CONCLUDING REMARKS

Our aim in this report was to demonstrate that the built-in hardware data structures of the DBC can emulate a network CODASYL-like data model in an effective manner. We also demonstrate the efficiency of the DBC in supporting CODASYL-like operations and systems. These studies are part of the overall goal of proving the efficacy of the DBC in supporting any existing data model and system. That the hierarchical data model can be efficiently emulated on the DBC has already been demonstrated in [1]. In a forthcoming report, we shall attempt to emulate the relational model and system.

The process of supporting an existing network database necessitates a one-time conversion of the database, to generate an equivalent DBC database. This process is straightforward. It is only necessary to represent each record, its area of occurrence and its existence in sets. This is easily accomplished by including all this information in the DBC record in the form of attribute-value pairs, called keywords. The DBC record is divested of all address-dependent pointers. However, in a conventional system pointers are also used to indicate the fact that a record belongs to a set and to indicate the position of the record within the set. With the elimination of pointers (in order to fully utilize the associative-searching capability of the DBC), this information is represented in the DBC by two keywords, one indicating the membership in (or ownership of) a set occurrence and the other showing the position through a sequence number. While loading the database, for each record stored, some keywords are designated as directory keywords. Directories are necessary since the database store is not a monolithic associative memory. Instead, only partitions are individually content-addressable. Each partition is called a minimal access unit (MAU) and corresponds to a disk cylinder with parallel read-out and associative search capability. Therefore, directories are maintained in order to minimize MAU accesses. Records are automatically clustered by designating certain directory keywords as clustering keywords. With proper clustering, all searches may be limited to only a few MAUs.

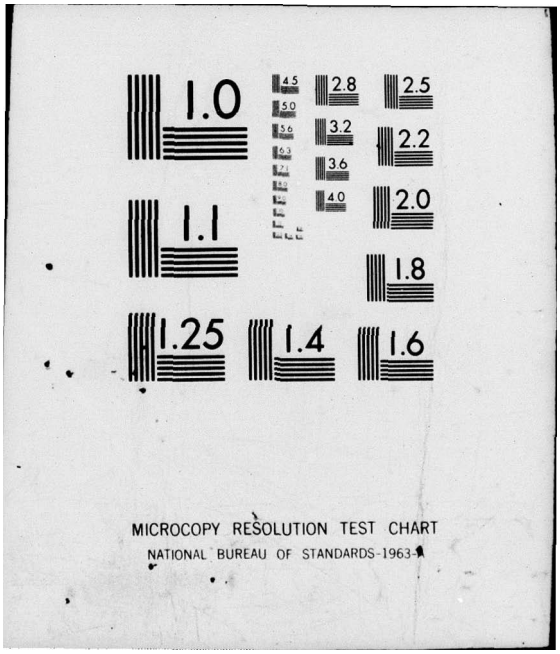
Records can also be automatically grouped by designating certain directory keywords as security keywords as well. Properly grouped records are said to form security atoms. It is important to note that records of a security atom have the same security requirements. Thus, directory keywords play the additional role of clustering keywords for performance enhancement and security keywords for access control. Because conventional CODASYL systems do not have any mechanism for protection by content, we did not elaborate this DBC

AD-A041 651 OHIO STATE UNIV COLUMBUS COMPUTER AND INFORMATION SC--ETC F/G 5/2
DBC SOFTWARE REQUIREMENTS FOR SUPPORTING NETWORK DATABASES.(U)
JUN 77 J BANERJEE , D K HSIAO, D S KERR N00014-75-C-0573

UNCLASSIFIED OSU-CISRC-TR-77-4 NL

2 OF 2
AD
A041851

END
DATE
FILMED
8-77


1.0
4.5
5.0
5.6
6.3
7.1
8.0
9.0
2.8
3.2
3.6
4.0
2.5
2.2
2.0
1.8
1.1
1.25
1.4
1.6
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

feature in the present report. Such features will be discussed in a future report.

Database conversion is done only once in an installation that already has a network database. The manufacturer of the DBC also supplies the interface module (DBTGI) which is a software package running in a front-end computer system to support user application programs. In Section 5, we have presented the design of this interface module. The interface module will intercept all storage, search and update commands issued by the user programs. It will then use the DBC to carry out all the database accesses necessary to execute these commands. Due to the simplicity of the interface software, its implementation is also easy. Users will have no knowledge of the type of computer system being used. They can still run their existing application programs and the only difference they will notice is a drastic reduction in response time. Furthermore, the freed-up main memory and CPU cycles will enhance the performance of the front-end computer.

In making a performance analysis, it has been observed that the mass memory requirement on the DBC is almost fifty percent more than that required by a conventional pointer-based implementation. However, this price in mass-memory storage is more than compensated by a large performance enhancement, by the removal of the conventional database management system software and by the saving of the real memory of the front-end computer. In fact, the average number of mass memory accesses required by the DBC is 10 to 100 times fewer than that required by a conventional system. Furthermore, the DBC directory size may be expected to be 100 times smaller than the size of the indexes of a conventional system.

There are other advantages as well in using the DBC. The DBTG model does not provide for accessing record occurrences by content, except through sets. Given any arbitrary data item values, it may be of considerable advantage to find all records of a particular type that contain these values, no matter what sets the records participate in (for example, finding all employee records, whose salary fields have a value greater than $10000, is independent of any consideration of sets). This type of command can be easily supported on the DBC because of its associative-search capability together with the policy of clustering by record type. In a conventional implementation, extensive directories will be necessary, and even then it will involve perhaps one page access for each record occurrence, since records are not clustered by their types. Another feature that can be easily included in a DBC-supported network model is the provision of security and protection mechanisms based on record

content. This feature is directly supported on the DBC. The DBC will associatively retrieve only those records that satisfy a given query and also satisfy the security specification, while a conventional system will first retrieve the page containing the required record and then check via software means whether the record violates security specifications or not.

We have shown how useful the DBC can be in emulating a network data model. However, the network model is definitely not the best data model to support on the DBC, even though it can be implemented more efficiently on the DBC than on a regular computer system. The network model provides for a sequential type of processing that was originally designed for a conventional location-addressing computer system. The DBC has far greater powers and in order to use it effectively, the data model must allow for a larger amount of parallel processing. Thus, with the advent of database machines, new data models will evolve that are much more powerful and flexible than existing models. Database computers will support these powerful models with an ease that can never be emulated by present-day computers. In addition, it must be noted that the performance gain in implementing existing data models on the DBC is further enhanced by automatic garbage collection (by hardware), large storage capability (of the order of 10^{10} bytes), automatic directory search as opposed to a search by software means, sophisticated security provisions (also automatic) and finally by high reliability since large and complex software systems can be replaced by specialized hardware functional modules.

References

[1] Hsiao, D.K., D.S. Kerr and F. Ng, "DBC software Requirements for Supporting Hierarchical Databases," The Ohio State University, Tech. Rep. No. OSU-CISRC-TR-77-1, April 1977.

[2] CODASYL Data Base Task Group, April 1971 Report, ACM, New York.

[3] Baum, R.I., D.K. Hsiao and K. Kannan, "The Architecture of a Database Computer - Part I : Concepts and Capabilities", The Ohio State University, Tech. Rep. No. OSU-CISRC-TR-76-1, September 1976.

[4] Hsiao, D.K. and K. Kannan, "The Architecture of a Database Computer - Part II : The Design of Structure Memory and its Related Processors", The Ohio State University, Tech. Rep. No. OSU-CISRC-TR-76-2, October 1976.

[5] Hsiao, D.K. and K. Kannan, "The Architecture of a Database Computer - Part III : The Design of the Mass Memory and its Related Components," The Ohio State University, Tech. Rep. No. OSU-CISRC-TR-76-3,December 1976.

[6] Date, C.J., An Introduction to Data Base Systems, Addison-Wesley Publishing Company, Reading, Massachussetts, 1975.

[7] Taylor, R.W. and R.L. Frank, "CODASYL Data-Base Management Systems," ACM Computing Surveys, Vol. 8, No. 1, March 1976, pp. 67-103.

[8] UNIVAC 1100 Series, Data Management System (DMS 1100) Schema Definition, Data Administrator Reference, Sperry Rand Corp., 1972, 1973.

[9] UNIVAC 1100 Series, Data Management System (DMS 1100) American National Standard COBOL (Fieldata), Data Manipulation Language, Programmer Reference, Sperry Rand Corp., 1972.